Contents

Benefits of this Library

The library was intended to satisfy the following primary goals:

- **Support parsing of all host and ipv4/ipv6 address formats in usage** plus some additional formats (see below or see javadoc for 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. All strings in the list below represent the same IPv4 subnet:
  - 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.
- **Polymorphism** in the code for IPv4/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
- **Sorting and comparison** of host names, addresses, address strings and subnets
- **Integrate with the java classes InetAddress, Inet6Address, Inet4Address.**

Supported 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 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
- * represents all addresses both ipv4 and ipv6
- /x which is just a network prefix length x with no associated address, is considered to be the mask for that network prefix length
- “” the empty string is considered the default loopback

For a more detailed list or formats parsed, 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. So 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

Core classes

The core classes are **`HostName`**, **`IPAddressString`**, and **`IPAddress`** along with the `IPAddress` subclasses **`IPv4Address`** and **`IPv6Address`**. If you have a textual representation of an address, then start with `HostName` or `IPAddressString`. If you have numeric bytes or integers, then start with `IPV4Address` or `IPV6Address` or `IPAddress.from`. Note that address instances can represent either a single address or a subnet.

Conversion from Textual Representation to Address

You can use one of:

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

or

```
try {
	IPAddress address = new IPAddressString("1.2.3.4").toAddress();
	...
} catch (IPAddressStringException 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]
```

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.

Host and Network Sections

Use `getHostSection()` and `getNetworkSection()` to get the host and network sections of an address.

```
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 can a section of an address, most of the same methods are available as those available with address instances.

Address Manipulations

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

The method `toSubnet` produces a subnet using a mask and/or a prefix. Note that this method applies the mask to each and every represented address. This means you can start with a subnet and end up with a single address or a smaller subnet, depending on the mask. One overloaded version of this method that takes both a prefix and mask will apply that prefix after the mask is applied, which can potentially increase subnet size.

The following code demonstrates how to get the lowest address in a prefixed subnet using either `getLowest` or using a mask.

```
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.toSubnet(mask);
System.out.println("address " + address + " masked " + maskedAddress);

IPAddress subnet = address.toSubnet(prefixLength);
IPAddress maskedSubnet = subnet.toSubnet(mask);
System.out.println("subnet " + subnet + " masked " + maskedSubnet);
IPAddress lowestAddressInSubnet = subnet.getLowest();
System.out.println("lowest in subnet " + lowestAddressInSubnet);
System.out.println(maskedAddress.equals(maskedSubnet));
System.out.println(maskedAddress.equals(lowestAddressInSubnet));
```

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
```

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 it 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 address with a prefix.
- `toMinimalPrefixed`: Similar to `toPrefixedEquivalent`, except that if the address already has a prefix, then `toPrefixedEquivalent` returns that address, whereas this method will try to reduce the prefix length if it can to match the range of values. When an address has no prefix, the two methods are equivalent.
- `toSupernet`: Will create a larger network by reducing the existing prefix with the given number of bits. Unlike `toSubnet`, this method will never reduce network size. It is similar to the `toSubnet` method that takes only a prefix, except that this always produces an address with a smaller prefix number.

Conversion to Textual Represention of Address

The `IPAddress` and `IPAddressSection` classes and their version-specific subclasses have many 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());
        System.out.println(address.toIPv6().toMixedString());
        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

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::

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:*::
```

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:%:%:%:%:%
```

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
```

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 Given 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.%.%
```

Conversion to Textual Representation for URL

Given a host name or address, use `HostName.toNormalizedString()` for the host part of a URL.

Subnet Membership

To check whether an 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
```

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 textual representation for a URL, use `HostName.toURLString()`.

Sorting and Comparison

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. 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`.