XBee Java Library

User Guide

Revision history—90001438

Revision	Date	Description
C	June 2015	Upgraded XBee Java Library to version v1.1.0: Added support for explicit frames and application layer fields. Added examples that demonstrate the new functionality of the API.
D	April 2016	Upgraded XBee Java Library to version v1.1.1: Added support for S2C 802.15.4 (XBee S1B), and added new unit tests.
E	January 2017	Upgraded XBee Java Library to version v1.2.0: Added support for XBee Cellular and XBee Wi-Fi protocols, compatibility with Android and new examples.
F	August 2017	Upgraded XBee Java Library to version v1.2.1: Added support for XBee Cellular NB-IoT and Thread protocols. Added IPv6 and CoAP support, as well as new examples and unit tests.

Trademarks and copyright

Disclaimers

Warranty

To view product warranty information, go to the following website:

www.digi.com/howtobuy/terms

Send comments

Documentation feedback: To provide feedback on this document, send your comments to techcomm@digi.com.

Customer support

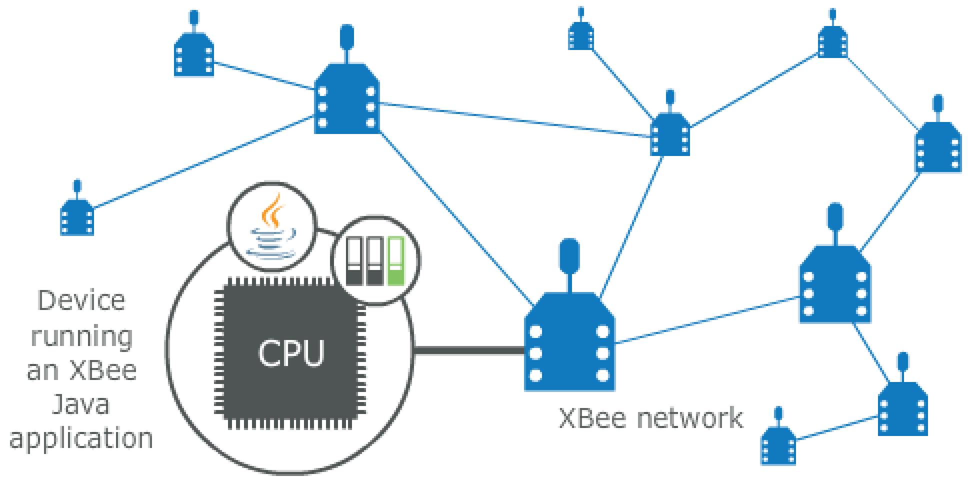
Digi Technical Support: Digi offers multiple technical support plans and service packages to help our customers get the most out of their Digi product. For information on Technical Support plans and pricing, contact us at +1 952.912.3444 or visit us at www.digi.com/support.

Contents

XBee Java Library

XBee devices allow you to enable wireless connectivity to your projects creating a network of connected devices. They provide features to exchange data with other devices in the network, configure them and control their I/O lines. An application running in an intelligent device can take advantage of these features to monitor and manage the entire network.

Despite the available documentation and configuration tools for working with XBee devices, it is not always easy to develop these kinds of applications.



The XBee Java Library is a Java API that dramatically reduces the time to market of XBee projects developed in Java and facilitates the development of these types of applications, making it an easy and smooth process. The XBee Java Library includes the following features:

- Support for multiple XBee devices and protocols.
- Support for the Android OS.
- High abstraction layer provides an easy-to-use workflow.
- Ability to configure local and remote XBee devices of the network.
- Discovery feature finds remote nodes on the same network as the local module.
- Ability to transmit and receive data from any XBee device on the network.
- Ability to manage the General Purpose Input and Output lines of all your XBee devices.

This portal provides the following documentation to help you with the different development stages of your Java applications using the XBee Java Library.

Getting started with XBee Java Library

This Getting Started Guide describes how to set up your environment and use the XBee Java Library to communicate with your XBee devices.

Start here to begin exploring the XBee Java Library. Then follow this guide to install the software, build and launch your first Java application, and begin communicating with your devices using Java.

Installing your software

The following software components are required to build and run your first XBee Java application:

- XBee Java Library software
- Java Virtual Machine
- XCTU
- Java IDE

XBee Java Library software

The first software package is the XBee Java Library. This package includes the XBee library, its source code and a collection of samples that will help you to develop Java applications to communicate with your XBee devices. You can download the latest version at:
https://github.com/digidotcom/XBeeJavaLibrary/releases

To work with this package, unzip the file you just downloaded. The main directory, *XBJL-X.Y.Z*, has the following structure:

- **/examples**: Several XBee Java Library examples to demonstrate the XBee Java Library features.
- **/extra-libs**: Libraries needed to build and launch an XBee Java application.
- **/javadoc**: XBee Java Library API documentation.
- **/src**: Source code for the XBee Java Library.
- **xbee-java-library-X.Y.Z.jar**: XBee Java Library jar file which allows you to easily interact with your XBee modules.
- **LICENSE.txt**: Legal licensing agreement.
- **README.md**
- **release_notes.txt**: Latest release information for XBee Java Library.

Java Virtual Machine

You must install a Java Virtual Machine to compile and launch Java projects. The recommended version is Java SE 8. If you already have a JRE or JDK 7 or higher installed on your PC, you can skip this step.

You can download the Java machine from
www.oracle.com/technetwork/java/javase/downloads/index.html

Once the download is complete, launch the program and follow the on-screen instructions to finish the installation process.

XCTU

Install XCTU 6.3.8 or later. XCTU is a free multi-platform application that enables developers to interact with Digi RF modules through a simple-to-use graphical interface. It includes new tools that make it easy to set up, configure, and test XBee RF modules.

For instructions on downloading and using XCTU, go to:

http://www.digi.com/xctu

Once you have downloaded XCTU, run the installer and follow the steps to finish the installation process.

After you load XCTU, a message about software updates appears. We recommend you always update XCTU to the latest available version.

Java IDE

To develop, build and launch Java applications, you can use an IDE (integrated development environment) capable of managing Java projects. There are many IDEs you can use, such as the following examples:

- Eclipse - http://www.eclipse.org
- NetBeans - https://netbeans.org
- Android Studio - https://developer.android.com/studio/

Configuring your XBee devices

You need to configure two XBee devices. One module (the sender) sends “Hello XBee World!” using the Java application. The other device (the receiver) receives the message.

Both devices must be working in the same protocol (802.15.4, Zigbee, DigiMesh or Point-to-Multipoint, or Wi-Fi) and must be configured to operate in the same network to enable communication.

Note If you are getting started with Cellular, you only need to configure one device. Cellular protocol devices are connected directly to the Internet, so there is not a network of remote devices to communicate with them. For the Cellular protocol, the XBee application demonstrated in the getting started guide differs from other protocols. The Cellular protocol sends and reads data from an echo server. All the steps of the guide but the the Add the application source code of the Building your first XBee Java application section are common to all the XBee devices regardless of their protocol.

Use XCTU to configure the devices. Plug the devices into the XBee adapters and connect them to your computer’s USB or serial ports.

Note For more information about XCTU, see the embedded help or see the XCTU User Guide. You can access the Help Contents from the Help menu of the tool.

Once XCTU is running, add your devices to the tool, and then select them from the **Radio Modules** section. When XCTU is finished reading the device parameters, complete the following steps, according to your device type.

Repeat these steps to configure all your XBee devices using XCTU.

- Add 802.15.4 devices
- Add Zigbee devices
- Add DigiMesh devices
- Add Point-to-Multipoint devices
- Add cellular devices
- Add Wi-Fi devices

Add 802.15.4 devices

1. Click **Load default firmware settings** in the Radio Configuration toolbar to load the default values for the device firmware.
2. Ensure the API mode (API1 or API2) is enabled. To do so, the **AP** parameter value must be **1** (API Mode Without Escapes) or 2 (API Mode With Escapes).
3. Configure **ID** (PAN ID) setting **CAFE**.
4. Configure **CH** (Channel setting) to **C**.
5. Click **Write radio settings** in the **Radio Configuration** toolbar to apply the new values to the module.
6. Once you have configured both modules, check to make sure they can see each other. Click **Discover radio modules in the same network** , the second button of the device panel in the **Radio Modules** view. The other device must be listed in the **Discovering remote devices** dialog.

Note If the other module is not listed, reboot both devices by pressing the **Reset** button of the carrier board and try adding the device again. If the list is still empty, go to the corresponding product manual for your devices.

Add Zigbee devices

1. For old Zigbee devices (S2 and S2B), ensure the devices are using **API firmware**. The firmware appears in the **Function** label of the device in the Radio Modules view.
   - One of the devices must be a coordinator - Function: Zigbee Coordinator API
   - We recommend the other one is a router - Function: Zigbee Router API

Note If any of the two previous conditions is not satisfied, you must change the firmware of the device. Click the **Update firmware** button of the Radio Configuration toolbar.

2. Click **Load default firmware settings** in the Radio Configuration toolbar to load the default values for the device firmware.
3. Do the following:
   - If the device has the **AP** parameter, set it to **1** (API Mode Without Escapes) or **2** (API Mode With Escapes).
   - If the device has the **CE** parameter, set it to **Enabled** in the coordinator.
4. Configure the PAN ID setting (**ID**) to be **C001BEE**.
5. Configure **SC** (Scan Channels) setting to **FFF**.
6. Click **Write radio settings** in the **Radio Configuration** toolbar to apply the new values to the module.

7. Once you have configured both modules, check to make sure they can see each other. Click **Discover radio modules in the same network**, the second button of the device panel in the **Radio Modules** view. The other device must be listed in the **Discovering remote devices** dialog.

Note If the other module is not listed, reboot both devices by pressing the **Reset** button of the carrier board and try adding the device again. If the list is still empty, go to the corresponding product manual for your devices.

Add DigiMesh devices

1. Click **Load default firmware settings** in the Radio Configuration toolbar to load the default values for the device firmware.
2. Ensure the API mode (API1 or API2) is enabled. The **AP** parameter value must be **1** (API Mode Without Escapes) or **2** (API Mode With Escapes).
3. Configure **ID** (PAN ID) setting to **CAFE**.
4. Configure **CH** (Operating Channel) setting to **C**.
5. Click **Write radio settings** in the **Radio Configuration** toolbar to apply the new values to the module.
6. Once you have configured both modules, check to make sure they can see each other. Click **Discover radio modules in the same network**, the second button of the device panel in the **Radio Modules** view. The other device must be listed in the **Discovering remote devices** dialog.

Note If the other module is not listed, reboot both devices by pressing the **Reset** button of the carrier board and try adding the device again. If the list is still empty, go to the corresponding product manual for your devices.

Add Point-to-Multipoint devices

1. Click **Load default firmware settings** in the Radio Configuration toolbar to load the default values for the device firmware.
2. Ensure the API mode (API1 or API2) is enabled. The **AP** parameter value must be **1** (API Mode Without Escapes) or **2** (API Mode With Escapes).
3. Configure **ID** (PAN ID) setting to **CAFE**.
4. Configure **HP** (Hopping Channel) setting to **5**.
5. Click **Write radio settings** in the **Radio Configuration** toolbar to apply the new values to the module.

6. Once you have configured both modules, check to make sure they can see each other. Click **Discover radio modules in the same network**, the second button of the device panel in the **Radio Modules** view. The other device must be listed in the **Discovering remote devices** dialog.

Note If the other module is not listed, reboot both devices by pressing the **Reset** button of the carrier board and try adding the device again. If the list is still empty, go to the corresponding product manual for your devices.

Add cellular devices

1. Click **Load default firmware settings** in the Radio Configuration toolbar to load the default values for the device firmware.
2. Ensure the API mode (API1 or API2) is enabled. To do so, the **AP** parameter value must be **1** (API Mode Without Escapes) or **2** (API Mode With Escapes).
3. Click **Write radio settings** in the Radio Configuration toolbar to apply the new values to the module.
4. Verify the module is correctly registered and connected to the Internet. To do so check that the LED on the development board blinks. If it is solid or has a double-blink, registration has not occurred properly. Registration can take several minutes.

Note In addition to the LED confirmation, you can check the IP address assigned to the module by reading the **MY** parameter and verifying it has a value different than **0.0.0.0**.

Add Wi-Fi devices

1. Click **Load default firmware settings** in the Radio Configuration toolbar to load the default values for the device firmware.
2. Ensure the API mode (API1 or API2) is enabled. To do so, the **AP** parameter value must be **1** (API Mode Without Escapes) or **2** (API Mode With Escapes).
3. Connect to an access point:
   a. Click the **Active Scan** button.
   b. Select the desired access point from the list of the **Active Scan** result dialog.
   c. If the access point requires a password, type your password.
   d. Click the **Connect** button and wait for the module to connect to the access point.
4. Click **Write radio settings** in the Radio Configuration toolbar to apply the new values to the module.
5. Verify the module is correctly connected to the access point by checking the IP address assigned to the module by reading the **MY** parameter and verifying it has a value different than **0.0.0.0**.

Building your first XBee Java application

In this section, you create and build your first XBee application. You can then use a device connected to your computer to broadcast the message “Hello XBee World!” to all remote devices on the same network using the XBee Java Library.

Note Cellular devices are connected directly to the Internet, so there is no network of remote devices to communicate with them. For the Cellular protocol, the XBee application demonstrated in this section differs from other protocols. The application sends and reads data from an echo server. All the steps in this section except the Add the application source code procedure are common to all the XBee devices, regardless of their protocol.

The following sections describe how to create and build the XBee application:

1. Create the project
2. Configure the project
3. Add the application source code
4. Build the application

The section describes the steps for the two most popular development environments: NetBeans and Eclipse. We also include instructions for building an application without using an IDE. You should be able to replicate these steps for a different Java IDE or any build automation tool.

Create the project

To use the XBee Java Library in your code, the first step is to create a new project to store the Java source code files and the build result. The name of the project is **myFirstXBeeApp**, and has the following structure:

- A directory called **src** for the sources organized in packages (**com.digi.xbee.example**).
- The **libs** folder to contain the XBee Java Library and other resources needed in order to properly build.
- The **bin** directory to store the ***.class** files that are the result of the build process.

To create the **myFirstXBeeApp** project, choose one of these development options and follow the steps:

- Eclipse
- Netbeans
- Command line

After you create the project, you must code the application and add the required libraries to the classpath of the project, because the classes with the functionality to communicate with your XBee devices are provided in a jar file (**xbee-java-library-X.Y.Z.jar**).

Eclipse

To create a new Java project in Eclipse, follow these steps:

1. Navigate to the **File** menu, select **New**, and click **Java Project**.

   A **New Java Project** window appears.

2. Enter the **Project name**, **myFirstXBeeApp**, and change the location if desired.
3. Click **Finish** to create the project. The window closes and the project is listed in the **Package Explorer** view at the left side of the IDE.

Netbeans

To create a new Java project in NetBeans, follow these steps:

1. Navigate to the **File** menu and select **New project...**.

   You are prompted with a **New Project** window.
2. In the **Categories** frame, select **Java** > **Java Application** on the right panel.
3. Click **Next**.
4. Enter the **Project Name**, **myFirstXBeeApp**, and the **Project Location**.
5. Clear the **Create Main Class** option. This will be created later.
6. Click **Finish** to create the project. The window closes and the project is listed in the **Projects** view at the left side of the IDE.

Command line

Note The command samples used in this guide are for Windows PCs. Linux and MacOS computers operate in a similar manner.

1. Create a directory to store the application source code and other resources, called **myFirstXBeeApp**, and go inside this new directory.

```
~> mkdir myFirstXBeeApp
~> cd myFirstXBeeApp
~\myFirstXBeeApp>
```

2. Then create a folder to store the source code, **src** .

```
~\myFirstXBeeApp> mkdir src
~\myFirstXBeeApp>
```

3. Inside the **src** directory, create the folders that represent the packages of the Java application, **com.digi.xbee.example**.

```
~\myFirstXBeeApp> cd src
~\myFirstXBeeApp\src> mkdir com\digi\xbee\example
~\myFirstXBeeApp\src\com\digi\xbee\example>
```

Configure the project

To build the project you have just created, you must add the needed JAR files to the classpath, and tell Java where to find the required native libraries when launching the application.

The **XBJL-X.Y.Z** you downloaded and unzipped (see XBee Java Library software) contains the library JAR file, **xbee-java-library-X.Y.Z.jar**, and other needed resources in the directory called **extra-libs**. The XBee Java Library depends on the following JAR files and native libraries:

- **rxtx-2.2.jar**: RXTX library that provides serial communication in Java.
- **slf4j-api-1.7.12.jar**: Simple Logging Facade for Java (SLF4J) for logging.

- **slf4j-nop-1.7.12.jar**: SLF4J binding for NOP, silently discarding all logging.
- RXTX native library that depends on your PC operating system and the installed Java Virtual Machine (as an example we are going to use 32-bit Windows).
- **android-sdk-5.1.1.jar**: Library that provides all the necessary classes to create content for Android.
- **android-sdk-addon-3.jar**: Digi SDK Add-on for Android, which allows you to create apps for Digi Embedded devices.

Configure the project - Eclipse

1. Click **File** > **New** > **Folder**, and create a directory called **libs** in the root of the project to create a directory.
2. Copy the **xbee-java-library-X.Y.Z.jar** file and the contents of the **extra-libs** directory from the **XBJL-X.Y.Z** folder to the **libs** directory.
3. From the **Package Explorer** view, right-click your sample project and go to **Properties**.
4. In the list of categories, go to **Java Build Path**, select the **Libraries** tab, and click the **Add JARs...** button.
5. In the **JAR Selection** window, go to the **myFirstXBeeApp** project and select only the following files from inside the **libs** folder:
   - **xbee-java-library-X.Y.Z.jar**
   - **rxtx-2.2.jar**
   - **slf4j-api-1.7.12.jar**
   - **slf4j-nop-1.7.12.jar**
   - **android-sdk-5.1.1.jar**
   - **android-sdk-addon-3.jar**

Ensure only the libraries listed above are added to your project.

6. Click **OK** to add the libraries.

Note You can optionally register the included API documentation and source code for the XBee Java Library to review classes and methods documentation within Eclipse.
Find the Javadoc in the installation directory, **XBJL-X.Y.Z**, inside **javadoc** directory, and the source code inside **src/main**.

7. Expand the **rxtx-2.2.jar** file of the **Libraries** tab list, select the **Native library location** item and click **Edit...**.
8. Select the **Workspace...** button to navigate to the **libs\native\Windows\win32** folder, and click **OK** to add the path to the native libraries.

Note The path to the native libraries depends on your computer operating system and the Java Virtual Machine you have installed (32-bit/64-bit).

9. Click **OK** to apply the Java Build Path property modifications.

Configure the project - Netbeans

1. Click **File** > **New** > **Folder**, and create a directory called **libs** in the root of the project to create a directory.
2. Copy the **xbee-java-library-X.Y.Z.jar** file and the contents of the **extra-libs** directory from the **XBJL-X.Y.Z** folder to the **libs** directory.
3. From **Projects** view, right-click your project and go to **Properties**.
4. In the list of categories, go to **Libraries** and click the **Add JAR/Folder** button.
5. In the **Add JAR/Folder** window, navigate to the myFirstXBeeApp project location, go to the libs directory, and select only the following files:
   - **xbee-java-library-X.Y.Z.jar**
   - **rxtx-2.2.jar**
   - **slf4j-api-1.7.12.jar**
   - **slf4j-nop-1.7.12.jar**
   - **android-sdk-5.1.1.jar**
   - **android-sdk-addon-3.jar**

Ensure only the libraries listed above are added to your project.

6. Click **Open** to finish.

Note You can optionally register the included API documentation and source code for the XBee Java Library to review classes and methods documentation within Eclipse.
Find the Javadoc in the installation directory, **XBJL-X.Y.Z**, inside **javadoc** directory, and the source code inside **src/main**.

7. Select **Run** in the left tree of the **Properties** dialog.
8. In the **VM Options** field, add the following option:

```
-Djava.library.path=libs\native\Windows\win32
```

The path is relative to the "myFirstXBeeApp's" path.

Note The path to the native libraries depends on your computer operating system and the Java Virtual Machine you have installed (32-bit/64-bit).

9. Click **OK** to apply the properties modifications.

Configure the project - Command line

You can specify all the resources required to build and launch the application in the command line. To facilitate that command, you can copy the needed resources and then define some environment variables.

1. Create a directory called **libs** in the root of the project.

```
~\myFirstXBeeApp> mkdir libs
~\myFirstXBeeApp>
```

2. Copy the **xbee-java-library-X.Y.Z.jar** file inside the **libs** directory.

```
~\myFirstXBeeApp> xcopy <path_to_XBJL>\XBJL-X.Y.Z\xbee-java-library-X.Y.Z.jar
libs
~\myFirstXBeeApp>
```

3. Copy the contents of the **extra-libs** directory in the **XBJL-X.Y.Z** folder to the **libs** directory.

```
~\myFirstXBeeApp> xcopy /S <path_to_XBJL>\XBJL-X.Y.Z\extra-libs libs

~\myFirstXBeeApp>
```

4. Define the following environment variables:

   XBJL_CLASS_PATH contains the paths to the required JAR files:

   - **xbee-java-library-X.Y.Z.jar**
   - **rxtx-2.2.jar**
   - **slf4j-api-1.7.12.jar**
   - **slf4j-nop-1.7.12.jar**
   - **android-sdk-5.1.1.jar**
   - **android-sdk-addon-3.jar**

```
~\myFirstXBeeApp> set XBJL_CLASS_PATH=libs\xbee-java-library-X.Y.Z.jar;libs\rxtx-
2.2.jar;libs\slf4j-api-1.7.12.jar;libs\slf4j-nop-1.7.12.jar
~\myFirstXBeeApp>
```

Add the application source code

Once you create your project, the next step is to create the Java source file to send the **Hello XBee World!** message to the rest of devices in the same network.

Add the application source code - Eclipse

1. In the Package Explorer view, select the project **myFirstXBeeApp** and right-click.
2. From the context menu, select **New** > **Class**. The **New Java Class** wizard opens.
3. Modify the **Package** to **com.digi.xbee.example**.
4. Type the **Name** of the class, **MainApp**.
5. Click **Finish**. Inside the **src** folder, a new package called **com.digi.xbee.example** is displayed, which contains the class MainApp you have just created.

   The **MainApp.java** file opens in the editor.

6. Remove the existing code and copy the appropriate source code from one the following links:
   - MainApp.java code (not cellular)
   - MainApp.java code (cellular)
7. Set the port (**PORT**) and baud rate (**BAUD_RATE**) of the module you are going to use as the sender in the code.

Note Use XCTU to find out the port and baud rate of your sender module. See the Frequently Asked Questions (FAQs) section for additional information.

8. Save the changes and close the file.

Add the application source code - Netbeans

1. In the **Projects** view, right-click and select the **myFirstXBeeApp project**.
2. From the context menu select **New** > **Java Class...**. The **New Java Class** wizard opens.
3. Modify the **Class Name** to **MainApp**.
4. Type the **Package** name, **com.digi.xbee.example**.
5. Click **Finish**. Inside the **Source Packages** folder, a new package, **com.digi.xbee.example**, is displayed, which contains the class MainApp you have just created.
   The **MainApp.java** file opens in the editor.
6. Remove the existing code and copy the appropriate source code from one the following links:
   - MainApp.java code (not cellular)
   - MainApp.java code (cellular)
7. Set the port (**PORT**) and baud rate (**BAUD_RATE**) of the module you are going to use as sender in the code.

Note Use XCTU to find out the port and baud rate of your sender module. See the Frequently Asked Questions (FAQs) section for additional information.

8. Save the changes and close the file.

Add the application source code - command line

1. Inside the last folder of the package structure you have just created (**myFirstXBeeApp/com/digi/xbee/example**), create the Java application source file, **MainApp.java**.

```
~\myFirstXBeeApp\com\digi\xbee\example> fsutil file createnew MainApp.java 0
File [...]\myFirstXBeeApp\com\digi xbee\example\MainApp.java is created
~\myFirstXBeeApp\com\digi\xbee\example>
```

2. Open the **MainApp.java** file in a text editor and copy the appropriate source code from one the following links:
   - MainApp.java code (not cellular)
   - MainApp.java code (cellular)

3. Set the port (**PORT**) and baud rate (**BAUD_RATE**) of the module you are going to use as the sender in the code.

Note Use XCTU to find out the port and baud rate of your sender module. See the Frequently Asked Questions (FAQs) section for additional information.

4. Save the changes and close the file.

MainApp.java code (not cellular)

This code must be modified to enter the right values for the constants **PORT** and **BAUD_RATE**. Their current values must be replaced with the port and baud rate of your sender module.

```
package com.digi.xbee.example;

import com.digi.xbee.api.WiFiDevice;
import com.digi.xbee.api.XBeeDevice;
import com.digi.xbee.api.exceptions.XBeeException;
import com.digi.xbee.api.models.XBeeProtocol;

public class MainApp {
    /* Constants */
    // TODO Replace with the port where your sender module is connected to.
    private static final String PORT = "COM1";
    // TODO Replace with the baud rate of your sender module.
    private static final int BAUD_RATE = 9600;

    private static final String DATA_TO_SEND = "Hello XBee World!";

    public static void main(String[] args) {
        XBeeDevice myDevice = new XBeeDevice(PORT, BAUD_RATE);
        byte[] dataToSend = DATA_TO_SEND.getBytes();

        try {
            myDevice.open();

            System.out.format("Sending broadcast data: '%s'", new String
(dataToSend));

            if (myDevice.getXBeeProtocol() == XBeeProtocol.XBEE_WIFI) {
                myDevice.close();
                myDevice = new WiFiDevice(PORT, BAUD_RATE);
                myDevice.open();
                ((WiFiDevice)myDevice).sendBroadcastIPData(0x2616, dataToSend);
            } else
                myDevice.sendBroadcastData(dataToSend);

            System.out.println(" >> Success");

        } catch (XBeeException e) {
            System.out.println(" >> Error");
            e.printStackTrace();
            System.exit(1);
        } finally {
            myDevice.close();
        }
    }
}
```

MainApp.java code (cellular)

This code must be modified to enter the right values for the constants **PORT** and **BAUD_RATE**. Their current values must be replaced with the port and baud rate of your sender module.

```
package com.digi.xbee.example;

import java.net.Inet4Address;
import java.net.UnknownHostException;

import com.digi.xbee.api.CellularDevice;
import com.digi.xbee.api.exceptions.XBeeException;
import com.digi.xbee.api.models.IPMessage;
import com.digi.xbee.api.models.IPProtocol;

public class MainApp {

    /* Constants */
    // TODO Replace with the serial port where your sender module is connected
to.
    private static final String PORT = "COM1";
    // TODO Replace with the baud rate of your sender module.
    private static final int BAUD_RATE = 9600;
    // TODO Optionally, replace with the text you want to send to the server.
    private static final String TEXT = "Hello XBee World!";

    private static final String ECHO_SERVER_IP = "52.43.121.77";
    private static final int ECHO_SERVER_PORT = 11001;

    private static final IPProtocol PROTOCOL_TCP = IPProtocol.TCP;

    public static void main(String[] args) {
        CellularDevice myDevice = new CellularDevice(PORT, BAUD_RATE);

        try {
            myDevice.open();

            System.out.format("Sending text to echo server: '%s'", TEXT);
            myDevice.sendIPData((Inet4Address) Inet4Address.getByName(ECHO_
SERVER_IP),
                    ECHO_SERVER_PORT, PROTOCOL_TCP, TEXT.getBytes());

            System.out.println(" >> Success");

            // Read the echoed data.
            IPMessage response = myDevice.readIPData();
            if (response == null) {
                System.out.format("Echo response was not received from the
server.");
                System.exit(1);
            }
            System.out.format("Echo response received: '%s'",
response.getDataString());
        } catch (XBeeException | UnknownHostException e) {
            System.out.println(" >> Error");
            e.printStackTrace();
            System.exit(1);
        } finally {
            myDevice.close();
        }
    }
}
```

Build the application

Now that you have configured the project and created the source code, you can build the example.

Build the application - Eclipse

Eclipse automatically builds all the projects by default, although this setting can be changed from the **Project** menu (**Build Automatically**).

If this setting is not enabled, you must manually build the project. Select your project from the **Package Explore** view. Once selected, go to the Project menu and click **Build project**.

Build the application - Netbeans

To build the project:

1. Select the project from the **Projects** view.
2. Right-click the project and select **Build**.

Build the application - Command line

The Java compiler, javac, needs to know the location of the source (-sourcepath), the required classes (-classpath), the output directory (-d) and the Java files to build (<source files>).

Compiling Java source with the javac command

```
javac \
       -sourcepath <path> \
       -classpath <path> \
       -d <directory> \
       <source files>
```

1. Create the **bin** directory to store the ***.class** files resulting from the Java build inside the folder **myFirstXBeeApp**:

```
~\myFirstXBeeApp> mkdir bin
~\myFirstXBeeApp>
```

2. Execute the following command, using the **XBJL_CLASS_PATH** variable created in the previous step:

```
~\myFirstXBeeApp> javac -sourcepath src -classpath %XBJL_CLASS_PATH% -d bin
src/com/digi/xbee/example/*.java
~\myFirstXBeeApp>
```

   Once the code is built, you obtain the class files (***.class**) inside the **bin** folder.

3. When the code is built, you can create an executable JAR file of your project.
   a. To create an executable JAR, create a manifest file which contains the main class and a reference to the required libraries.

   For this example, the **manifest.mf** file is located inside the **root** project folder.

Example of manifest.mf

```
        Main-Class: com.digi.xbee.example.MainApp
        Class-Path:libs/xbee-java-library-X.Y.Z.jar libs/rxtx-2.2.jar libs/slf4j-api-
1.7.7.jar libs/slf4j-nop-1.7.7.jar
```

Note The last line of the manifest file must be an empty line.

b. When you have the manifest file, create the executable JAR file.

Creating an executable JAR file with the jar command

```
jar \
       cvfm \
       <jar file> \
       <manifest file> \
       <files>
```

You must execute this command from the **bin** folder as shown in the following example:

```
~\myFirstXBeeApp\bin> jar cvfm myFirstXBeeApp.jar ..\manifest.mf .
~\myFirstXBeeApp\bin>
```

Launching the application

After you have built the project, you can launch it. Your application needs the XBee Java Library and the required JAR files as well as the native code to properly run your application.

- Launching the application for non-Cellular protocol
- Launching the application for Cellular protocol

Launching the application for non-Cellular protocol

If you developed the application for protocols other than Cellular, you must ensure the message *Hello XBee World!* is sent from your sender device. Use XCTU to read the frames received:

1. Launch XCTU.
2. Add the receiver module to XCTU. Do not use the same module you established as the sender in the Java Application.
3. Click **Open the serial connection with the radio module** to switch to **Consoles working mode** and open the serial connection. This allows you to see the data when it is received.
4. Launch the Java application using the following command line, Eclipse or NetBeans instructions.
   - Eclipse
   - Netbeans
   - Command line

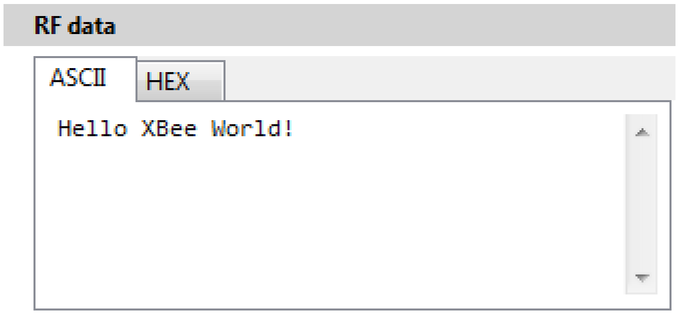
The application sends the message *Hello XBee World!* to all modules of the network. When that happens, a line with the result of the operation prints to the standard output:

```
Sending broadcast data: 'Hello XBee World!' >> Success
```

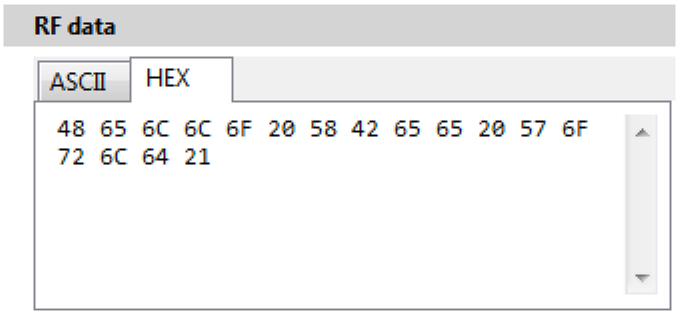
Verify that a new RX frame or RX IPv4 (if the module is Wi-Fi) appears in the XCTU console. Select the frame and review the details as shown in the following example:

Start delimiter	7E
Length	Depends on the XBee protocol
Frame type	Depends on the XBee protocol
64-bit source address	XBee sender's 64-bit address
RF data/Received data	48 65 6C 6C 6F 20 58 42 65 65 20 57 6F 72 6C 64 21

Where the sequence in *RF data/Received data* is the hexadecimal representation of the sent ASCII string: **Hello XBee World!**



```
Hello XBee World!
```



```
48 65 6C 6C 6F 20 58 42 65 65 20 57 6F
72 6C 64 21
```

Launching the application for Cellular protocol

If you developed the application for the Cellular protocol, complete the following tasks:

- Execute the application.
- Verify the data sent to the echo server echoes back and the Cellular module reads it correctly.

Launch the Java application using the command line, Eclipse or NetBeans instructions below.

- Eclipse
- Netbeans
- Command line

The application sends the message *Hello XBee World!* to the echo server and the Cellular device reads it back. When that happens, a line with the result of the operation is printed to the standard output:

```
Sending text to echo server: 'Hello XBee World!' >> Success
Echo response received: 'Hello XBee World!'
```

Using the XBee Java Library

This section of the guide provides reference information for the XBee Java Library. In addition to serving as a reference, it provides detailed information about the additional capabilities of this product.

The XBee Java Library is an easy-to-use API developed in Java that allows you to interact with Digi's XBee radio frequency (RF) modules. You can use the XBee Java Library to create any kind of Java or Android application, from command line to GUI, that needs to communicate with or configure XBee devices.

The API is designed both for new and advanced users. You do not need previous knowledge of XBee communication protocols or advanced Java experience to get started. The API provides all the methods you need to perform the most common tasks related to XBee devices. If you are an advanced user, you can take advantage of the complete set of API commands to create powerful applications.

The XBee Java Library includes the following features:

- Support for the following XBee devices:
  - Zigbee
  - 802.15.4
  - DigiMesh
  - Point-to-Multipoint
  - Wi-Fi
  - Cellular
  - Cellular NB-IoT
  - Thread
- Support for API and API Escaped operating modes.
- Support for Android.

- A range of capabilities, including the ability to:
    - Discover all the remote XBee devices in your network.
    - Configure your XBee device or any remote module of the network.
    - Send data to a specific device, or to all the XBee devices in the network.
    - Receive data from remote XBee devices.
    - Receive network status changes related to your XBee device.
    - Configure, set and read the IO lines of your XBee devices.
    - Receive IO data samples at a specific rate from any remote XBee device in the network.

Before you begin to work with the XBee Java Library, we recommended looking at the concepts explained in the first section, XBee terminology, to help you while developing your application.

XBee terminology

This section covers basic XBee concepts and terminology. The XBee Java library manual refers to these concepts frequently, so it is important to understand these concepts.

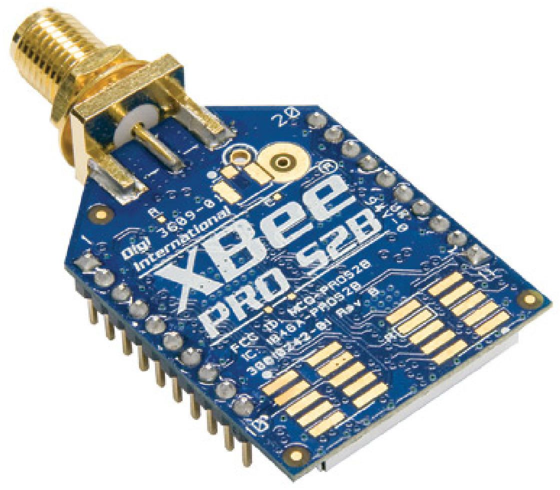
RF modules

A radio frequency (RF) module is a small electronic circuit used to transmit and receive radio signals on different frequencies. Digi produces a wide variety of RF modules to meet the requirements of almost any wireless solution, such as long-range, low-cost, and low power modules. The most popular wireless products are the XBee RF modules.

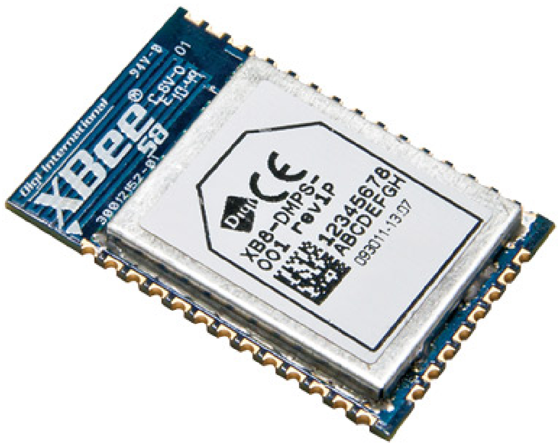
XBee RF modules

XBee is the brand name of a family of RF modules produced by Digi International Inc. XBee RF modules are modular products that make it easy and cost-effective to deploy wireless technology. Multiple protocols and RF features are available, giving customers enormous flexibility to choose the best technology for their needs.

The XBee RF modules are available in two form-factors: Through-Hole and Surface Mount, with different antenna options. Almost all modules are available in the Through-Hole form factor and share the same footprint.



XBee Through-Hole (THT)



XBee Surface Mount (SMT)

Radio firmware

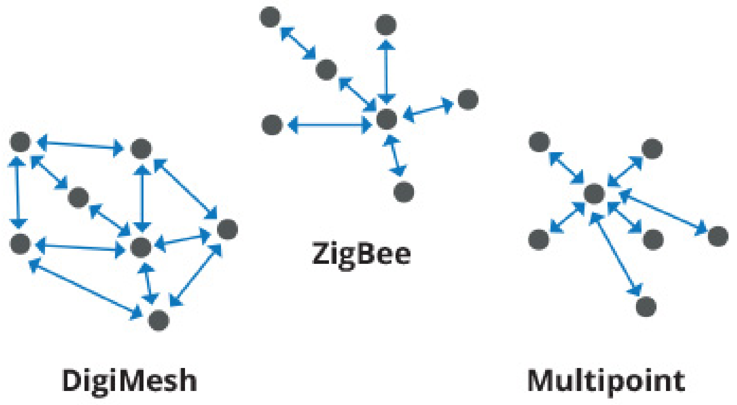
Radio firmware is the program code stored in the radio module's persistent memory that provides the control program for the device. From the local web interface of the XBee Gateway, you can update or change the firmware of the local XBee module or any other module connected to the same network. This is a common task when changing the role of the device or updating to the latest version of the firmware.

Radio communication protocols

A radio communication protocol is a set of rules for data exchange between radio devices. An XBee module supports a specific radio communication protocol depending on the module and its radio firmware.

Following is the complete list of protocols supported by the XBee radio modules:

- IEEE 802.15.4
- Zigbee
- Zigbee Smart Energy
- DigiMesh (Digi's proprietary)
- ZNet
- IEEE 802.11 (Wi-Fi)
- Point-to-multipoint (Digi's proprietary)
- XSC (XStream compatibility)
- Cellular
- Cellular NB-IoT
- Thread



Note Not all XBee devices can run all these communication protocols. The combination of XBee hardware and radio firmware determines the protocol that an XBee device can execute. Refer to the XBee RF Family Comparison Matrix for more information about the available XBee RF modules and the protocols they support.

Radio module operating modes

The operating mode of an XBee radio module establishes the way a user or any microcontroller attached to the XBee communicates with the module through the Universal Asynchronous Receiver/Transmitter (UART) or serial interface.

Depending on the firmware and its configuration, the radio modules can work in three different operating modes:

- Application Transparent (AT) operating mode
- API operating mode
- API escaped operating mode

In some cases, the operating mode of a radio module is established by the firmware version and the firmware's AP setting. The module's firmware version determines whether the operating mode is AT or API. The firmware's AP setting determines if the API mode is escaped (**AP**=2) or not (**AP**=1). In other cases, the operating mode is only determined by the AP setting, which allows you to configure the mode to be AT (**AP**=0), API (**AP**=1) or API escaped (**AP**=2).

API operating mode

Application Programming Interface (API) operating mode is an alternative to AT operating mode. API operating mode requires that communication with the module through a structured interface; that is, data communicated in API frames.

The API specifies how commands, command responses, the module sends and receives status messages using the serial interface. API operation mode enables many operations, such as the following:

- Configure the XBee device itself.
- Configure remote devices in the network.
- Manage data transmission to multiple destinations.
- Receive success/failure status of each transmitted RF packet.
- Identify the source address of each received packet.

Depending on the AP parameter value, the device can operate in one of two modes: API (**AP** = **1**) or API escaped (**AP** = **2**) operating mode.

Application Transparent (AT) operating mode

In Application Transparent (AT) or transparent operating mode, all serial data received by the radio module is queued up for RF transmission. When the module receives RF data, it sends the data out through the serial interface.

To configure an XBee module operating in AT, put the device in command mode to send the configuration commands.

Command mode

When the radio module is working in AT operating mode, configure settings using the command mode interface.

To enter command mode, send the 3-character command sequence through the serial interface of the radio module, usually **+++**, within one second. Once the command mode has been established, the module sends the reply **OK**, the command mode timer starts, and the radio module can receive AT commands.

The structure of an AT command follows this format:

```
AT[ASCII command][Space (optional)][Parameter (optional)][Carriage return]
```

Example:

```
ATNI MyDevice\r
```

If no valid AT commands are received within the command mode timeout, the radio module automatically exits command mode. You can also exit command mode issuing the **CN** command (Exit Command mode).

API escaped operating mode

API escaped operating mode (**AP** = 2) works similarly to API mode. The only difference is that when working in API escaped mode, some bytes of the API frame specific data must be escaped.

Use API escaped operating mode to add reliability to the RF transmission, which prevents conflicts with special characters such as the start-of-frame byte (0x7E). Since 0x7E can only appear at the start of an API packet, if 0x7E is received at any time, you can assume that a new packet has started regardless of length. In API escaped mode, those special bytes are escaped.

Escape characters

When sending or receiving an API frame in API escaped mode, you must escape (flag) specific data values so they do not interfere with the data frame sequence. To escape a data byte, insert 0x7D and follow it with the byte being escaped, XOR'd with 0x20.

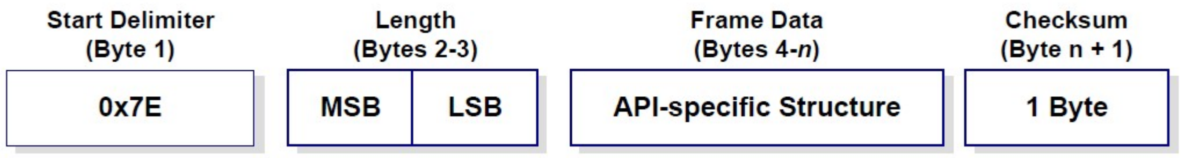
The following data bytes must be escaped:

- 0x7E: Frame delimiter
- 0x7D: Escape
- 0x11: XON
- 0x13: XOFF

API frames

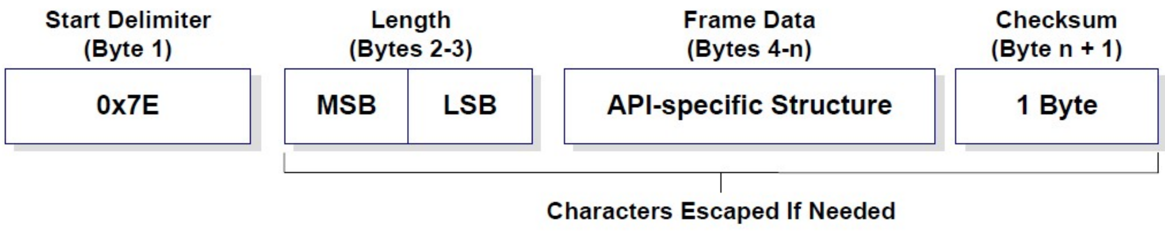
An API frame is the structured data sent and received through the serial interface of the radio module when it is configured in API or API escaped operating modes. API frames are used to communicate with the module or with other modules in the network.

An API frame has the following structure:



Start Delimiter	This field is always 0x7E.
Length	The length field has a two-byte value that specifies the number of bytes that are contained in the frame data field. It does not include the checksum field.
Frame Data	The content of this field is composed by the API identifier and the API identifier specific data. Depending on the API identifier (also called API frame type), the content of the specific data changes.
Checksum	Byte containing the hash sum of the API frame bytes.

In API escaped mode, there may be some bytes in the Length, Frame Data and Checksum fields that must be escaped.



AT settings or commands

The firmware running in the XBee RF modules contains a group of settings and commands that you can configure to change the behavior of the module or to perform any related action. Depending on the protocol, the number of settings and meanings vary, but all the XBee RF modules can be configured with AT commands.

All the firmware settings or commands are identified with two ASCII characters and some applications and documents refer to them as **AT settings** or **AT commands**.

The configuration process of the AT settings varies depending on the operating mode of the XBee RF module.

- AT operating mode. In this mode, you must put the module in a special mode called command mode, so it can receive AT commands. For more information about configuring XBee RF modules working in AT operating mode, see Application Transparent (AT) operating mode.
- API operating mode. To configure or execute AT commands when the XBee RF module operates in API mode, you must generate an AT command API frame containing the AT setting identifier and the value of that setting, and send it to the XBee RF module. For more information about API frames, see API frames.

Working with XBee classes

When working with the XBee Java Library, start with an XBee device object that represents a physical module. A physical XBee device is the combination of hardware and firmware. Depending on that combination, the device runs a specific wireless communication protocol such as Zigbee, 802.15.4, DigiMesh, Wif-Fi, and Cellular. An XBeeDevice class represents the XBee module in the API.

Most of the protocols share the same features and settings, but there are some differences between them. For that reason, the XBee Java Library also includes a set of classes that represent XBee devices running different communication protocols. The XBee Java Library supports one XBee device class per protocol, as follows:

- XBee Zigbee device (**ZigbeeDevice**)
- XBee 802.15.4 device (**Raw802Device**)
- XBee DigiMesh device (**DigiMeshDevice**)
- XBee Point-to-multipoint device (**DigiPointDevice**)

- XBee IP devices
    - XBee Cellular device (**CellularDevice**)
        - XBee Cellular NB-IoT device (**NBIoTDevice**)
    - XBee Wi-Fi device (**WiFiDevice**)
- XBee IPv6 devices
    - XBee Thread device (**ThreadDevice**)

All these XBee device classes allow you to configure the physical XBee device, communicate with the device, send data to other nodes on the network, receive data from remote devices, and so on. Depending on the class, you may have additional methods to execute protocol-specific features or similar methods.

To work with the API and perform actions involving the physical device, you must instantiate a generic XBeeDevice object or one that is protocol-specific. This documentation refers to the XBeeDevice object generically when describing the different features, but they can be applicable to any XBee device class.

This section provides information to help you complete the following tasks:

- Instantiate an XBee device object
- Open the XBee device connection
- Close the XBee device connection

Instantiate an XBee device object

When you are working with the XBee Java Library, the first step is to instantiate an XBee device object. The API works well using the generic XBeeDevice class, but you can also instantiate a protocol-specific XBee device object if you know the protocol your physical XBee device is running.

An XBee device is represented as either **local** or **remote** in the XBee Java Library, depending upon how you communicate with the device.

Local XBee device

A local XBee device is the object in the library representing the device that is physically attached to your PC through a serial or USB port. The classes you can instantiate to represent a local device are listed in the following table:

Class	Description
XBeeDevice	Generic object, protocol independent
ZigbeeDevice	Zigbee protocol
Raw802Device	802.15.4 protocol
DigiMeshDevice	DigiMesh protocol
DigiPointDevice	Point-to-multipoint protocol
CellularDevice	Cellular protocol
WiFiDevice	Wi-Fi protocol

Class	Description
NBIoTDevice	Cellular NB-IoT protocol
ThreadDevice	Thread protocol

To instantiate a generic or protocol-specific XBee device, you need to provide the following two parameters:

- Serial port name
- Serial port baud rate

Instantiating a local XBee device - simple

```
import com.digi.xbee.api.XBeeDevice;

[...]

// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600);
```

Other serial port parameters are optional and they default to the following values:

Data bytes	8
Stop bits	1
Parity	None
Flow control	None

There are also other constructors allowing their specification.

Instantiating a local XBee device - advanced

```
import com.digi.xbee.api.XBeeDevice;

[...]

// Instantiate an XBee device with optional serial params.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600, 8, 1, 0, 0);
```

The XBee Java API also includes a serial port configuration class that you can use to declare an XBee device.

Instantiating a local XBee device - serial port parameters

```
import com.digi.xbee.api.XBeeDevice;
import com.digi.xbee.api.connection.serial.SerialPortParameters;

[...]

// Instantiate an XBee device using the SerialPortParameters class.
SerialPortParameters portParams = new SerialPortParameters(
                        9600, /* baudrate:     9600 */
                        8,    /* data bits:    8 */
                        1,    /* stop bits:    1 */
                        0,    /* parity:       none */
                        0,    /* flow control: none */);
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", portParams);
```

Remote XBee device

Remote XBee device objects represent remote nodes of the network. These are XBee devices that are not attached to your PC but operate in the same network as the attached (local) device.

When working with remote XBee devices, it is very important to understand that you cannot communicate directly with them. You need to provide a local XBee device operating in the same network that acts as bridge between your serial port and the remote node.

Managing remote devices is similar to managing local devices, but with limitations. You can configure them, handle their IO lines, and so on, in the same way you manage local devices. Local XBee devices have several methods for sending data to remote devices, but the remote devices cannot use these methods because they are already remote. Therefore a remote device cannot send data to another remote device.

In the local XBee device instantiation you can choose between instantiating a generic remote XBee device object, or a protocol-specific remote XBee device. The following table lists the remote XBee device classes:

Class	Description
RemoteXBeeDevice	Generic object, protocol independent
RemoteZigbeeDevice	Zigbee protocol
RemoteRaw802Device	802.15.4 protocol
RemoteDigiMeshDevice	DigiMesh protocol
RemoteDigiPointDevice	Point-to-multipoint protocol
RemoteThreadDevice	Thread protocol

Note XBee Cellular and Wi-Fi protocols do not support remote devices.

To instantiate a remote XBee device object, you need to provide the following parameters:

- Local XBee device attached to your PC that serves as the communication interface.
- 64-bit address of the remote device (for all remote devices but RemoteThreadDevice).
- IPv6 address of the remote device (only if the remote device is a RemoteThreadDevice).

RemoteRaw802 device objects can be also instantiated by providing the local XBee device attached to your PC and the 16-bit address of the remote device.

Instantiating a remote XBee device

```
import com.digi.xbee.api.RemoteXBeeDevice;
import com.digi.xbee.api.XBeeDevice;
```

```
[...]

// Instantiate a local XBee device object.
XBeeDevice myLocalXBeeDevice = new XBeeDevice("COM1", 9600);

// Instantiate a remote XBee device object.
RemoteXBeeDevice myRemoteXBeeDevice = new RemoteXBeeDevice(myLocalXBeeDevice,
                                  new XBee64BitAddress("000000409D5EXXXX"));
```

The local device must also be the same protocol for protocol-specific remote XBee devices.

Open the XBee device connection

Before trying to communicate with the local XBee device attached to your PC, you need to open its communication interface, which is typically a serial/USB port. Use the **open()** method of the instantiated XBee device, and you can then communicate and configure the device.

Remote XBee devices do not have an open method. They use a local XBee device as the connection interface. If you want to perform any operation with a remote XBee device you must open the connection of the associated local device.

If the connection is not open, any task executed by the XBee device object involving communication with the physical device throws an **InterfaceNotOpenException** runtime exception, terminating the execution of your application. Similarly, if you try to open a device that was already opened, you receive an **InterfaceAlreadyOpenException** runtime exception and your application exits. This is a common issue if you are working with remote XBee devices.

Open the device connection

```
import com.digi.xbee.api.XBeeDevice;

[...]

// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600);


// Open the device connection.
myXBeeDevice.open();

[...]
```

The **open()** method may fail for the following reasons:

- All the possible errors are caught as **XBeeException**:
  - If the connection interface is already in use by other applications, throwing an **InterfaceInUseException**.
  - If the interface is invalid or does not exist, throwing an **InvalidInterfaceException**.
  - If the configuration used to open the interface is not valid, throwing an **InvalidConfigurationException**.
  - If you do not have permissions to open the interface, throwing a **PermissionDeniedException**.
  - If the operating mode of the device is not API or API_ESCAPE, throwing an **InvalidOperatingModeException**.

The open() action performs some other operations apart from opening the connection interface of the device. It reads the device information (reads some sensitive data from it) and determines the operating mode of the device.

Device information reading

The read device information process reads the following parameters from the local or remote XBee device and stores them inside. You can then access parameters at any time, calling their corresponding getters.

- 64-bit address
- 16-bit address
- Node Identifier
- Firmware version
- Hardware version
- IPv4 address (only for Cellular and Wi-Fi modules)
- IPv6 address (only for Thread modules)
- IMEI (only for Cellular modules)

The read process is performed automatically in local XBee devices when opening them with the **open ()** method. If remote XBee devices cannot be opened, you must use **readDeviceInfo()** to read their device information.

Initializing a remote XBee device

```
import com.digi.xbee.api.RemoteXBeeDevice;
import com.digi.xbee.api.XBeeDevice;
import com.digi.xbee.api.models.XBee64BitAddress;

[...]

// Instantiate an XBee device object.
XBeeDevice myLocalXBeeDevice = new XBeeDevice("COM1", 9600);
myLocalXBeeDevice.open();

// Instantiate a remote XBee device object.
RemoteXBeeDevice myRemoteXBeeDevice = new RemoteXBeeDevice(myLocalXBeeDevice,
                                  new XBee64BitAddress("0013A20040XXXXXX"));

// Read the device information of the remote XBee device.
myRemoteXBeeDevice.readDeviceInfo();

[...]
```

The **readDeviceInfo()** method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a **TimeoutException**.

- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

Note Although the **readDeviceInfo** method is executed automatically in local XBee devices when they are open, you can issue it at any time to refresh the information of the device.

Getting the device information

```
import com.digi.xbee.api.XBeeDevice;

[...]

XBeeDevice myXBeeDevice = ...

myXBeeDevice.open();

// Get the 64-bit address of the device.
XBee64BitAddress 64BitAddress = myXBeeDevice.get64BitAddress();
// Get the Node identifier of the device.
String nodeIdentifier = myXBeeDevice.getNodeID();
// Get the Hardware version of the device.
HardwareVersion hardwareVersion = myXBeeDevice.getHardwareVersion();
// Get the Firmware version of the device.
String firmwareVersion = myXBeeDevice.getFirmwareVersion();
```

The read device information process also determines the communication protocol of the local or remote XBee device object. This is typically something you need to know beforehand if you are not using the generic **XBeeDevice object**.

However, the API performs this operation to ensure that the class you instantiated is the correct one. So, if you instantiated a Zigbee device and the **open()** process realizes that the physical device is actually a DigiMesh device, you receive an **XBeeDeviceException** indicating the device.

Getting the XBee protocol

You can retrieve the protocol of the XBee device from the object executing the corresponding getter.

```
import com.digi.xbee.api.XBeeDevice;

[...]

XBeeDevice myXBeeDevice = ...

myXBeeDevice.open();

// Get the protocol of the device.
XBeeProtocol xbeeProtocol = myXBeeDevice.getXBeeProtocol();
```

Device operating mode

The **open()** process also reads the operating mode of the physical local device and stores it in the object. As with previous settings, you can retrieve the operating mode from the object at any time by calling the corresponding getter.

Getting the operating mode

```
import com.digi.xbee.api.XBeeDevice;

[...]

XBeeDevice myXBeeDevice = ...

myXBeeDevice.open();

// Get the operating mode of the device.
OperatingMode operatingMode = myXBeeDevice.getOperatingMode();
```

Remote devices do not have an **open()** method, so you receive **UNKNOWN** when retrieving the operating mode of a remote XBee device.

The XBee Java API supports 2 operating modes for local devices:

- API
- API with escaped characters

This means that AT (transparent) mode is not supported by the API. So, if you try to execute the **open ()** method in a local device working in AT mode, you get an **XBeeException** caused by an **InvalidOperatingModeException**.

Close the XBee device connection

Once you have finished communicating with the local XBee device, we recommend that you close its communication interface. This releases the interface so other applications can use it.

To close the connection of a local XBee device, use the **close()** method of the XBee device object. This method immediately frees the allocated resources.

```
[...]

// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = ...;

try {
    myXBeeDevice.open();
} catch [...] {
    [...]
} finally {
    // Close the device connection.
    myXBeeDevice.close();
}

[...]
```

Remote XBee devices cannot be open, so they cannot be closed either. To close the connection of a remote device you need to close the connection of the local associated device.

Configuring the XBee device

One of the main features of the XBee Java Library is the ability to configure the parameters of local and remote XBee devices and execute actions or commands on them.

The values set on the different parameters are not persistent through subsequent resets unless you store those changes in the device. For more information, see Write configuration changes.

Read and set common parameters

Local and remote XBee device objects provide a set of methods to get and set common parameters of the device. Some of these parameters are saved inside the XBee device object, and a cached value is returned when the parameter is requested. Other parameters are read directly from the physical XBee device when requested.

Cached parameters

There are some parameters in an XBee device that are used or requested frequently. To avoid the overhead of those parameters being read from the physical XBee device every time they are requested, they are saved inside the XBeeDevice object being returned when the getters are called.

The following table lists parameters that are cached and their corresponding getters:

Parameter	Method
64-bit address	**get64BitAddress()**
16-bit address	**get16BitAddress()**
Node identifier	**getNodeIdentifier()**
Firmware version	**getFirmwareVersion()**
Hardware version	**getHardwareVersion()**

Local XBee devices read and save previous parameters automatically when opening the connection of the device. In remote XBee devices, you must issue the **readDeviceInfo()** method to initialize the parameters.

You can refresh the value of those parameters (that is, read their values and update them inside the XBee device object) at any time by calling the **readDeviceInfo()** method.

Refreshing cached values

```
import com.digi.xbee.api.XBeeDevice;

[...]

// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600);
myXBeeDevice.open();


// Refresh the cached values.
myXBeeDevice.readDeviceInfo();

[...]
```

The **readDeviceInfo()** method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

All the cached parameters but the Node Identifier do not change; therefore, they cannot be set. For the Node Identifier, there is a method within all the XBee device classes that allows you to change it:

Method	Descripton
setNodeIdentifier (String)	Specifies the new Node Identifier of the device. This method configures the physical XBee device with the provided Node Identifier and updates the cached value with the one provided.

Non-cached parameters

The following are the non-cached parameters that have their own methods to be configured within the XBee device classes:

- **Destination Address**: This setting specifies the default 64-bit destination address of a module that is used to report data generated by the XBee device (that is, IO sampling data). This setting can be get and set.

Method	Description
getDestinationAddress()	Returns the XBee64BitAddress of the device where the data will be reported.
setDestinationAddress (XBee64BitAddress)	Specifies the 64-bit address of the device where the data will be reported. Configures the destination address of the XBee device with the one provided.

- **PAN ID**: This is the ID of the Personal Area Network the XBee device is operating in. This setting can be get and set.

Method	Description
getPANID()	Returns a byte array containing the ID of the Personal Area Network where the XBee device is operating.
setPANID (byte[])	Specifies the 64-bit value in a byte array format of the PAN ID where the XBee device should work.

- **Power level**: This setting specifies the output power level of the XBee device. This setting can be get and set.

Method	Description
getPowerLevel()	Returns a **PowerLevel** enumeration entry indicating the power level of the XBee device.
setPowerLevel (PowerLevel)	Specifies a **PowerLevel** enumeration entry containing the desired output level of the XBee device.

Configuring non-cached parameters

```
import com.digi.xbee.api.XBeeDevice;

[...]

// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600);
myXBeeDevice.open();

// Set the destination address of the device.
XBee64BitAddress destinationAddress = new XBee64BitAddress("0123456789ABCDEF");
myXBeeDevice.setDestinationAddress(destinationAddress);

// Read the operating PAN ID of the device.
byte[] operatingPANID = myXBeeDevice.getPANID();

// Read the output power level.
PowerLevel powerLevel = myXBeeDevice.getPowerLevel();

[...]
```

All the previous getters and setters of the different options may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

Common Parameters Example

The XBee Java Library includes a sample application that displays how to get and set common parameters. It can be located in the following path:

/examples/configuration/ManageCommonParametersSample

Read, set and execute other parameters

If you want to read or set a parameter that does not have a custom getter or setter within the XBee device object, you can do so. All the XBee device classes (local or remote) include two methods to get and set any AT parameter, and a third one to run a command in the XBee device.

Getting a parameter

You can read the value of any parameter of an XBee device using the **getParameter()** method provided by all the XBee device classes. Use this method to get the value of a parameter that does not have its getter method within the XBee device object.

Method	Descripton
getParameter (String)	Specifies the AT command (string format) to retrieve its value. The method returns the value of the parameter in a byte array.

Getting a parameter from the XBee device

```
import com.digi.xbee.api.XBeeDevice;

[...]

// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600)
myXBeeDevice.open();

// Get the value of the Sleep Time (SP) parameter.
byte[] sleepTime = myXBeeDevice.getParameter("SP");

[...]
```

The **getParameter()** method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

Set and get parameters example

The XBee Java Library includes a sample application that displays how to get and set parameters using the methods explained previously. It can be located in the following path:

/examples/configuration/SetAndGetParametersSample

Setting a parameter

To set a parameter that does not have its own setter method, you can use the **setParameter()** method provided by all the XBee device classes.

Method	Descripton
setParameter (String, byte[])	Specifies the AT command (String format) to be set in the device and a byte array containing the value of the parameter.

Setting a parameter in the XBee device

```
import com.digi.xbee.api.XBeeDevice;

[...]

// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600);
myXBeeDevice.open();

// Configure the Node ID using the setParameter method.
myXBeeDevice.setParameter("NI", "YODA".getBytes());

[...]
```

The **setParameter()** method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

Set and get parameters example

The XBee Java Library includes a sample application that displays how to get and set parameters using the methods explained previously. It can be located in the following path:

/examples/configuration/SetAndGetParametersSample

Executing a command

There are other AT parameters that cannot be read or written. They are actions that are executed by the XBee device. The XBee library has several commands that handle most common executable parameters, but to run a parameter that does not have a custom command, you can use the **executeCommand()** method provided by all the XBee device classes.

Method	Descripton
executeCommand(String)	Specifies the AT command (String format) to be run in the device.

Running a command in the XBee device

```
import com.digi.xbee.api.XBeeDevice;

[...]

// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600);
myXBeeDevice.open();

// Run the apply changes command.
myXBeeDevice.executeCommand("AC");
```

```
[...]
```

The executeCommand() method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

Apply configuration changes

By default, when you perform any configuration on a local or remote XBee device, the changes are automatically applied. However, there could be some scenarios when you want to configure different settings or parameters of a device and apply the changes at the end when everything is configured. For that purpose, the XBeeDevice and RemoteXBeeDevice objects provide some methods that allow you to manage when to apply configuration changes.

Method	Description	Notes
enableApplyConfigurationChanges (boolean)	Specifies whether the changes on settings and parameters are applied when set.	The apply configuration changes flag is enabled by default.
isApplyConfigurationChangesEnabled ()	Returns whether the XBee device is configured to apply parameter changes when they are set.	
applyChanges()	Applies the changes on parameters that were already set but are pending to be applied.	This method is useful when the XBee device is configured to not apply changes when they are set.

Applying configuration changes

```
import com.digi.xbee.api.XBeeDevice;

[...]

// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600);
myXBeeDevice.open();

// Check if device is configured to apply changes.
boolean applyChangesEnabled = myXBeeDevice.isApplyConfigurationChangesEnabled();

// Configure the device not to apply parameter changes automatically.
```

```
if (applyChangesEnabled)
myXBeeDevice.setApplyConfigurationChanges(false);

// Set the PAN ID of the XBee device to BABE.
myXBeeDevice.setPANID(new byte[]{(byte)0xBA, (byte)0xBE});

// Perform other configurations.
[...]

// Apply changes.
myXBeeDevice.applyChanges();

[...]
```

The applyChanges() method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

Write configuration changes

If you want configuration changes performed in an XBee device to persist through subsequent resets, you need to write those changes in the device. Writing changes means that the parameter values configured in the device are written to the non-volatile memory of the XBee device. The module loads the parameter values from non-volatile memory every time it is started.

The XBee device classes (local and remote) provide a method to write (save) the parameter modifications in the XBee device memory so they persist through subsequent resets: **writeChanges()**.

Writing configuration changes

```
import com.digi.xbee.api.XBeeDevice;

[...]

// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600);
myXBeeDevice.open();

// Set the PAN ID of the XBee device to BABE.
myXBeeDevice.setPANID(new byte[]{(byte)0xBA, (byte)0xBE});

// Perform other configurations.
[...]

// Apply changes.
myXBeeDevice.applyChanges()

// Write changes.
myXBeeDevice.writeChanges()
```

```
[...]
```

The **writeChanges()** method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

Reset the device

There are times when it is necessary to reset the XBee device because things are not operating properly or you are initializing the system. All the XBee device classes of the XBee API provide the **reset()** method to perform a software reset on the local or remote XBee module.

In local modules, the **reset()** method blocks until a confirmation from the module is received, which usually takes one or two seconds. Remote modules do not send any kind of confirmation, so the method does not block when resetting them.

Resetting the module

```
import com.digi.xbee.api.XBeeDevice;

[...]

// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600);
myXBeeDevice.open();

// Reset the module.
myXBeeDevice.reset();

[...]
```

The reset() method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

Reset example

The XBee Java Library includes a sample application that shows you how to perform a reset on your XBee device. The example is located in the following path:

/examples/configuration/ResetModuleSample

Configure Wi-Fi settings

Unlike other protocols such as Zigbee or DigiMesh where devices are connected each other, the XBee Wi-Fi protocol requires that the module is connected to an access point in order to communicate with other TCP/IP devices.

This configuration and connection with access points can be done using applications such as XCTU; however, the XBee Java Library includes a set of methods to configure the network settings, scan access points and connect to one of them in easily.

Example: Configure Wi-Fi settings and connect to an access point

The XBee Java Library includes a sample application that demonstrates how to configure the network settings of a Wi-Fi device and connect to an access point. You can locate the example in the following path:

/examples/configuration/ConnectToAccessPointSample

Configure IP addressing mode

Before connecting your Wi-Fi module to an access point, you must decide how to configure the network settings using the IP addressing mode option. The supported IP addressing modes are contained in an enumerator called **IPAddressingMode**. It allows you to choose between:

- **DHCP**
- **STATIC**

The method used to perform this configuration is:

Method	Description
setIPAddressingMode (IPAddressingMode)	Sets the IP addressing mode of the Wi-Fi module. Depending on the provided mode, network settings are configured differently: ▪ **DHCP**. Network settings are assigned by a server. ▪ **STATIC**. Network settings must be provided manually one by one.

Configuring IP addressing mode

```
import com.digi.xbee.api.wiFiDevice;
import com.digi.xbee.api.models.IPAddressingMode;

[...]

// Instantiate a Wi-Fi device object.
WiFiDevice myWiFiDevice = new WiFiDevice("COM1", 9600);
myWiFiDevice.open();

// Configure the IP addressing mode to DHCP.
myWiFiDevice.setIPAddressingMode(IPAddressingMode.DHCP);

// Save the IP addressing mode.
myWiFiDevice.writeChanges();


[...]
```

The **setIPAddressingMode(IPAddressingMode)** method may fail for the following reasons:

- There is a timeout setting the IP addressing parameter, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
    - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
    - The response of the command is not valid, throwing an **ATCommandException**.
    - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

Configure IP network settings

Like any TCP/IP protocol device, the XBee Wi-Fi modules have the IP address, subnet mask, default gateway and DNS settings that you can get at any time using the XBee Java Library.

Unlike some general configuration settings, these parameters are not saved inside the WiFiDevice object. Every time you request the parameters, they are read directly from the Wi-Fi module connected to the computer. The following is the list of parameters used in the configuration of the TCP/IP protocol:

Parameter	Method
IP Address	**getIPAddress()**
Subnet mask	**getIPAddressMask()**
Gateway IP	**getGatewayIPAddress()**
DNS Address	**getDNSAddress()**

Configuring IP network settings

```
import com.digi.xbee.api.wiFiDevice;
import com.digi.xbee.api.models.IPAddressingMode;

[...]

// Instantiate a Wi-Fi device object.
WiFiDevice myWiFiDevice = new WiFiDevice("COM1", 9600);
myWiFiDevice.open();

// Configure the IP addressing mode to DHCP.
myWiFiDevice.setIPAddressingMode(IPAddressingMode.DHCP);

// Connect to access point with SSID 'My SSID' and password "myPassword".
myWiFiDevice.connect("My SSID", "myPassword");

// Display the IP network settings that were assigned by the DHCP server.
System.out.println("- IP address: " + myWiFiDevice.getIPAddress().getHostAddress
());
System.out.println("- Subnet mask: " + myWiFiDevice.getIPAddressMask
().getHostAddress());
System.out.println("- Gateway IP address: " + myWiFiDevice.getGatewayIPAddress
().getHostAddress());
System.out.println("- DNS IP address: " + myWiFiDevice.getDNSAddress
().getHostAddress());

[...]
```

Any of the previous methods may fail for the following reasons:

Parameter	Method
IP Address	**setIPAddress(Inet4Address)**
Subnet mask	**setIPAddressMask(Inet4Address)**
Gateway IP	**setGatewayIPAddress(Inet4Address)**
DNS Address	**setDNSAddress(Inet4Address)**

Configuring IP network settings

```
import java.net.Inet4Address;
import com.digi.xbee.api.wiFiDevice;
import com.digi.xbee.api.models.IPAddressingMode;

[...]

// Instantiate a Wi-Fi device object.
WiFiDevice myWiFiDevice = new WiFiDevice("COM1", 9600);
myWiFiDevice.open();

// Configure the IP addressing mode to Static.
myWiFiDevice.setIPAddressingMode(IPAddressingMode.STATIC);

// Configure the IP network settings.
myWiFiDevice.setIPAddress((Inet4Address)Inet4Address.getByName("192.168.1.123"));
myWiFiDevice.setIPAddressMask((Inet4Address)Inet4Address.getByName
("255.255.255.0"));
myWiFiDevice.setGatewayIPAddress((Inet4Address)Inet4Address.getByName
("192.168.1.1"));
myWiFiDevice.setDNSAddress((Inet4Address)Inet4Address.getByName("8.8.8.8"));

// Save the IP network settings.
myWiFiDevice.writeChanges();

// Connect to access point with SSID 'My SSID' and password "myPassword"
myWiFiDevice.connect("My SSID", "myPassword");

[...]
```

Any of the previous methods may fail for the following reasons:

- There is a timeout setting the IP addressing parameter, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

These set methods that configure the network settings can be only invoked when the IP addressing mode is **Static**, otherwise an **XBeeException** appears.

Scan and connect to access points

The XBee Java Library includes some helpful methods in the WiFiDevice class to scan and connect to access points. The AccessPoint class represents an access point in the XBee Java Library and contains the following:

- The SSID of the access point.
- The encryption type of the access point represented by a value of the **WiFiEncryptionType** enumerator, including:
  - NONE
  - WPA
  - WPA2
  - WEP
- The channel where the access point operates.
- The signal quality with the device in %.

Although you can instantiate an AccessPoint object in your code, they are usually generated and returned by the access point scan methods of the **WiFiDevice**.

Scanning for access points

In order to scan access points, a method within the **WiFiDevice** returns a list of **AccessPoint** objects found in the vicinity:

Method	Description
scanAccessPoints()	Performs a scan to search for access points in the vicinity. Returns a list with the access points found.

Scanning for access points

```
import com.digi.xbee.api.wiFiDevice;
import com.digi.xbee.api.models.AccessPoint;

[...]

// Instantiate a Wi-Fi device object.
WiFiDevice myWiFiDevice = new WiFiDevice("COM1", 9600);
myWiFiDevice.open();

// Scan for access points.
List<AccessPoint> accessPoints = myWiFiDevice.scanAccessPoints();

// Print information of the access points found:
System.out.println("Access points found:");
for (AccessPoint accessPoint:accessPoints)
System.out.println("  - " + accessPoint.toString());

[...]
```

The **scanAccessPoints()** method may fail for the following reasons:

- There is a timeout setting the IP addressing parameter, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

Note This method blocks until the scan process ends or the access point timeout expires. This timeout is set to 15 seconds by default, but you can configure it using the **getAccessPointTimeout** and **setAccessPointTimeout** methods of the **WiFiDevice** class.

Getting/setting the access point operations timeout

```
import com.digi.xbee.api.WiFiDevice;

[...]

public static final int NEW_TIMEOUT_FOR_AP_OPERATIONS = 30 * 1000; // 30 seconds

WiFiDevice myWiFiDevice =

[...]

// Retrieve the configured timeout for access point related operations.
System.out.println("Current access point timeout: " +
myWiFiDevice.getAccessPointTimeout() + " ms.");

[...]

// Configure the new access point related operations timeout (in milliseconds).
myWiFiDevice.setAccessPointTimeout(NEW_TIMEOUT_FOR_AP_OPERATIONS);

[...]
```

If you already know the SSID of the access point you want to get, you can use the getAccessPoint (String) method to get it. If the access point with the provided SSID is found, it is returned as an AccessPoint object.

Method	Description
getAccessPoint(String)	Finds and reports the access point that matches the supplied SSID.

Getting an access point with specific SSID

```
import com.digi.xbee.api.wiFiDevice
import com.digi.xbee.api.models.AccessPoint;

[...]

// Instantiate a Wi-Fi device object.
WiFiDevice myWiFiDevice = new WiFiDevice("COM1", 9600);
```

```
myWiFiDevice.open();

// Get the access point with SSID "My access point".
AccessPoint accessPoint = myWiFiDevice.getAccessPoint("My access point");

[...]
```

Note This method blocks until the scan process ends or the access point timeout expires. This timeout is set to 15 seconds by default, but you can configure it using the **getAccessPointTimeout** and **setAccessPointTimeout** methods of the **WiFiDevice** class.

Connecting to an access point

Once you have found the access point you want to connect to, you can use any of the connect methods provided by the **WiFiDevice** object to connect. The connect methods require the password of the access point as a parameter. If the **WiFiEncryptionType** of the access point is **NONE**, the password you provide must be null. In any case, the connect methods return a boolean value indicating if the connection was established successfully.

Method	Description
connect (AccessPoint, String)	Connects to the provided access point.
connect(String, String)	Connects to the access point with the provided SSID. If you already know the SSID this method allows you to skip the scan step.

Connecting to an access point

```
import com.digi.xbee.api.wiFiDevice;
import com.digi.xbee.api.models.AccessPoint;

[...]

// Instantiate a Wi-Fi device object.
WiFiDevice myWiFiDevice = new WiFiDevice("COM1", 9600);
myWiFiDevice.open();

// Get the access point with SSID "My access point".
AccessPoint accessPoint = myWiFiDevice.getAccessPoint("My access point");

// Connect to the access point.
boolean connected = myWiFiDevice.connect(accessPoint, "myPassword");

if (connected)
    System.out.println("Connected");
else
    System.out.println("Could not connect");
```

```

[...]
```

The connect methods may fail for the following reasons:

- There is a timeout setting the IP addressing parameter, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

Note These methods block until the device is fully connected to the access point or the access point timeout expires. This timeout is set to **15 seconds** by default, but you can configure it using the **getAccessPointTimeout** and **setAccessPointTimeout** methods of the **WiFiDevice** class.

Disconnecting from an access point

If you want to close the connection with the current access point to connect to a different access point or just disconnect from the network, call the disconnect method.

Method	Description
disconnect()	Disconnects from the access point where the device is connected.

Disconnecting from an access point

```
import com.digi.xbee.api.wiFiDevice;

[...]

// Instantiate a Wi-Fi device object.
WiFiDevice myWiFiDevice = new WiFiDevice("COM1", 9600);
myWiFiDevice.open();

// Get the access point with SSID "My access point".
myDevice.connect("My access point", "myPassword");

[...]

myDevice.disconnect();

[...]
```

The **disconnect()** method may fail for the following reasons:

- There is a timeout setting the IP addressing parameter, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

Note This method blocks until the device is fully disconnected from the access point or the access point timeout expires. This timeout is set to **15 seconds** by default, but you can configure it using the **getAccessPointTimeout** and **setAccessPointTimeout** methods of the **WiFiDevice** class.

Checking connection status

The **WiFiDevice** object provides a method that allows you to check the connection status (connected or disconnected) of your device at any time.

Method	Description
isConnected()	Returns whether the device is connected to an access point or not.

Checking connection status

```
import com.digi.xbee.api.wiFiDevice;

[...]

// Instantiate a Wi-Fi device object.
WiFiDevice myWiFiDevice = new WiFiDevice("COM1", 9600);
myWiFiDevice.open();


// Connect to the access point with SSID "My access point".
myWiFiDevice.connect("My access point", "myPassword");

// Check connection status.
System.out.println("Connected: " + myWiFiDevice.isConnected());

myWiFiDevice.disconnect();

// Check connection status again.
System.out.println("Connected: " + myWiFiDevice.isConnected());

[...]
```

The **isConnected()** method may fail for the following reasons:

- There is a timeout setting the IP addressing parameter, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

Getting the connected access point

If you need to get the access point where the device is connected, you can do so by executing the **getConnectedAccessPoint** method of the **WiFiDevice** object.

Method	Description
getConnectedAccessPoint()	Returns the access point where the Wi-Fi device is connected.

Getting the connected access point

```
import com.digi.xbee.api.wiFiDevice;
import com.digi.xbee.api.models.AccessPoint;

[...]

// Instantiate a Wi-Fi device object.
WiFiDevice myWiFiDevice = new WiFiDevice("COM1", 9600);
myWiFiDevice.open();

// Connect to the access point with SSID "My access point".
myWiFiDevice.connect("My access point", "myPassword");

// Get the connected access point.
AccessPoint connectedAccessPoint = myWiFiDevice.getConnectedAccessPoint();

// Print access point information.
System.out.println("SSID :" + connectedAccessPoint.getSSID());
System.out.println("Encryption :" + connectedAccessPoint.getEncryptionType());
System.out.println("Channel :" + connectedAccessPoint.getChannel());
System.out.println("Signal quality :" + connectedAccessPoint.getSignalQuality());

[...]
```

The **getConnectedAccessPoint()** method may fail for the following reasons:

- There is a timeout setting the IP addressing parameter, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

Discovering the XBee network

Several XBee modules working together and communicating with each other form a network. XBee networks have different topologies and behaviors depending on the protocol of the XBee devices that form the network.

The XBee Java Library includes a class, called XBeeNetwork, that represents the set of nodes forming the actual XBee network. This class allows you to perform some operations related to the nodes. The XBee Network object can be retrieved from a local XBee device after you open the device using the **getNetwork()** method.

Note Because XBee Cellular and Wi-Fi modules protocols are directly connected to the Internet and do not share a connection, these protocols do not support XBee networks.

Retrieving the XBee network

```
import com.digi.xbee.api.XBeeDevice;
import com.digi.xbee.api.XBeeNetwork;

[...]
```

```
// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600);
myXBeeDevice.open();

// Get the XBee Network object from the XBee device.
XBeeNetwork network = myXBeeDevice.getNetwork();

[...]
```

One of the main features of the XBeeNetwork class is the ability to discover the XBee devices that form the network. The XBeeNetwork object provides the following operations related to the XBee devices discovery feature:

- Configure the discovery process
- Discover the network
- Access the discovered devices
- Add and remove devices manually

Configure the discovery process

Before discovering all the nodes of a network you need to configure the settings of that process. The API provides two methods to configure the discovery timeout and discovery options. These methods set the provided values in the module.

Method	Description
setDiscoveryTimeout (long)	Configures the discovery timeout (NT parameter) with the given value in milliseconds.
setDiscoveryOptions (Set<DiscoveryOptions>)	Configures the discovery options (NO parameter) with the given set of options. The set of discovery options contains the different **DiscoveryOption** configuration values that are applied to the local XBee module when performing the discovery process. These options are the following: ■ **DiscoveryOption.APPEND_DD**: Appends the device type identifier (DD) to the information retrieved when a node is discovered. This option is valid for DigiMesh, Point-to-multipoint (Digi Point) and Zigbee protocols. ■ **DiscoveryOption.DISCOVER_MYSELF**: The local XBee device is returned as a discovered device. This option is valid for all protocols. ■ **DiscoveryOption.APPEND_RSSI**: Appends the RSSI value of the last hop to the information retrieved when a node is discovered. This option is valid for DigiMesh and Point-to-multipoint (Digi Point) protocols.

Configuring the timeout options

```
import com.digi.xbee.api.XBeeDevice;
import com.digi.xbee.api.XBeeNetwork;
import com.digi.xbee.api.models.DiscoveryOptions;

[...]

// Get the XBee Network object from the XBee device.
XBeeNetwork network = myXBeeDevice.getNetwork();

// Set the timeout to 10 seconds.
network.setDiscoveryTimeout(10000);

// Append the device type identifier and the local device.
network.setDiscoveryOptions(EnumSet.of(DiscoveryOptions.APPEND_DD,
DiscoveryOptions.DISCOVER_MYSELF));

[...]
```

Discover the network

The XBeeNetwork object discovery process allows you to discover and store all the XBee devices that form the network. The XBeeNetwork object provides a method for executing the discovery process:

Method	Description
startDiscoveryProcess ()	Starts the discovery process, saving the remote XBee devices found inside the XBeeNetwork object.

When a discovery process has started, you can monitor and manage it using the following methods provided by the XBeeNetwork object:

Method	Description
isDiscoveryRunning()	Returns whether the discovery process is running.
stopDiscoveryProcess()	Stops the discovery process that is taking place.

Although you call the **stopDiscoveryProcess** method, DigiMesh and DigiPoint devices are blocked until the configured discovery time has elapsed. If you try to get or set any parameter during that time, a **TimeoutException** is thrown.

Discovering the network

Once the process has finished, you can retrieve the list of devices that form the network using the **getDevices()** method provided by the network object.

```
import com.digi.xbee.api.XBeeDevice;
import com.digi.xbee.api.XBeeNetwork;

[...]
```

```
// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600);
myXBeeDevice.open();

// Get the XBee Network object from the XBee device.
XBeeNetwork network = myXBeeDevice.getNetwork();

// Start the discovery process.
network.startDiscoveryProcess();

// Wait until the discovery process has finished.
while (network.isDiscoveryRunning()) {
// Sleep.
}

// Retrieve the devices that form the network.
List<RemoteXBeeDevice> remotes = network.getDevices();

[...]
```

Click one of the following links to view the discovery methods:

- Discovering the network with a listener
- IDiscoveryListener implementation example, MyDiscoveryListener
- Removing the discovery listener
- Device discovery example

Discovering the network with a listener

The API also allows you to add a discovery listener to notify you when new devices are discovered, the process finishes, or an error occurs during the process. In this case, you need to provide a listener before starting the discovery process using the **addDiscoveryListener(IDiscoveryListener)** method.

```
import com.digi.xbee.api.XBeeDevice;
import com.digi.xbee.api.XBeeNetwork;
[...]

// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600);
myXBeeDevice.open();

// Get the XBee Network object from the XBee device.
XBeeNetwork network = myXBeeDevice.getNetwork();

// Create the discovery listener.
MyDiscoveryListener myDiscoveryListener = ...

// Add the discovery listener.
network.addDiscoveryListener(myDiscoveryListener);

// Start the discovery process.
network.startDiscoveryProcess();

[...]
```

IDiscoveryListener implementation example, MyDiscoveryListener

```
import com.digi.xbee.api.RemoteXBeeDevice;
import com.digi.xbee.api.listeners.IDiscoveryListener;

public class MyDiscoveryListener implements IDiscoveryListener {

       /*
        * Device discovered callback.
        */
        @Override
        public void deviceDiscovered(RemoteXBeeDevice discoveredDevice) {
               System.out.println("New device discovered: " +
                       discoveredDevice.toString());
       }

       /*
        * Discovery error callback.
        */
        @Override
        public void discoveryError(String error) {
               System.out.println("There was an error during the discovery: " +
                       error);
       }

       /*
        * Discovery finished callback.
        */
        @Override
        public void discoveryFinished(String error) {
               if (error != null)
                   System.out.println("Discovery finished due to an error: " +
                          error);
               else
                   System.out.println("Discovery finished successfully.");
       }
}
```

Removing the discovery listener

To remove the registered discovery listener, use the **removeDiscoveryListener(IDiscoveryListener)** method.

```
[...]

MyDiscoveryListener myDiscoveryListener = ...
network.addDiscoveryListener(myDiscoveryListener);

[...]

// Remove the discovery listener.
network.removeDiscoveryListener(myDiscoveryListener);

[...]
```

Device Discovery Example

The XBee Java Library includes a sample application that displays how to perform a device discovery using a listener. It can be located in the following path:

/examples/network/DiscoverDevicesSample

Discover specific devices

The XBeeNetwork object also provides a methods to discover specific devices of the network. This is useful, for example, if you only need to work with a particular remote device.

Method	Description
discoverDevice (String)	Specifies the node identifier of the XBee device to be found. Returns the remote XBee device whose node identifier equals the one provided. In the case of finding more than one device, it returns the first one.
discoverDevices (List<String>)	Specifies the node identifiers of the XBee devices to be found. Returns a list with the remote XBee devices whose node identifiers equal those provided.

Note These are blocking methods, so the application blocks until the devices are found or the configured timeout expires.

```
import com.digi.xbee.api.RemoteXBeeDevice;
import com.digi.xbee.api.XBeeDevice;
import com.digi.xbee.api.XBeeNetwork;

[...]

// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600);
myXBeeDevice.open();

// Get the XBee Network object from the XBee device.
XBeeNetwork network = myXBeeDevice.getNetwork();

// Discover the remote device whose node ID is ‘Yoda’.
RemoteXBeeDevice discoveredDevice = network.discoverDevice("Yoda");

ArrayList<String> ids = new ArrayList<String>();
ids.add("R2D2");
ids.add("C3PO");

// Discover the remote devices whose node IDs are ‘R2D2’ and ‘C3PO’.
ArrayList<RemoteXBeeDevice> discoveredDevices = network.discoverDevices(ids);

[...]
```

IDiscoveryListener implementation example, MyDiscoveryListener

MyDiscoveryListener must implement the **IDiscoveryListener** interface, which includes the methods that are executed when discover events occur.

The behavior of the listener is as follows:

- When a new remote XBee device is discovered, the **deviceDiscovered()** method of the **IDiscoveryListener** executes, providing the reference of the RemoteXBeeDevice discovered as a parameter. It is a reference, because the XBee network already stores that device inside its list of remote XBee devices.
- If there is an error during the discovery process, the **discoveryError()** method of the **IDiscoveryListener** executes, providing an error message with the cause of that error.

- When the discovery process finishes or the configured timeout expires, the **discoveryFinished ()** method of the **IDiscoveryListener** executes, providing the error message with the reason the process did not finish successfully, or null if the process finished successfully.

IDiscoveryListener implementation example, MyDiscoveryListener

```
import com.digi.xbee.api.RemoteXBeeDevice;
import com.digi.xbee.api.listeners.IDiscoveryListener;

public class MyDiscoveryListener implements IDiscoveryListener {

       /*
        * Device discovered callback.
        */
        @Override
        public void deviceDiscovered(RemoteXBeeDevice discoveredDevice) {
               System.out.println("New device discovered: " +
                       discoveredDevice.toString());
       }

       /*
        * Discovery error callback.
        */
        @Override
        public void discoveryError(String error) {
               System.out.println("There was an error during the discovery: " +
                       error);
       }

       /*
        * Discovery finished callback.
        */
        @Override
        public void discoveryFinished(String error) {
               if (error != null)
                   System.out.println("Discovery finished due to an error: " +
                          error);
               else
                   System.out.println("Discovery finished successfully.");
       }
}
```

Removing the discovery listener

To remove the registered discovery listener, use the **removeDiscoveryListener(IDiscoveryListener)** method.

```
[...]

MyDiscoveryListener myDiscoveryListener = ...
network.addDiscoveryListener(myDiscoveryListener);

[...]

// Remove the discovery listener.
network.removeDiscoveryListener(myDiscoveryListener);
```

```
[...]
```

Device discovery example

The XBee Java Library includes a sample application that displays how to perform a device discovery using a listener. It can be located in the following path:

/examples/network/DiscoverDevicesSample

Access the discovered devices

Once a discovery process has finished, the nodes discovered are saved inside the XBeeNetwork object. This means that you can get the devices stored inside at any time. Using the **getNumberOfDevices()** method you determine the number of devices found before getting them.

The following table contains a list of methods provided by the XBeeNetwork object that allow you to retrieve already discovered devices:

Method	Description
getDevices()	Returns the list of remote XBee devices.
getDevices(String)	Specifies the node identifier of the remote XBee devices to get from the network. Returns a list with the remote XBee devices whose node identifiers match the one specified.
getDevice(String)	Specifies the node identifier of the remote XBee device to get from the network. Returns the remote XBee device whose node identifier matches the one specified.
getDevice (XBee16BitAddress)	Specifies the 16-bit address of the remote XBee device to get from the network. Returns the remote XBee device whose 16-bit address matches the one specified.
getDevice (XBee64BitAddress)	Specifies the 64-bit address of the remote XBee device to be get from the network. Returns the remote XBee device whose 64-bit address matches the one specified.

Getting stored devices from the XBee network

```
import com.digi.xbee.api.RemoteXBeeDevice;
import com.digi.xbee.api.XBeeDevice;
import com.digi.xbee.api.XBeeNetwork;
import com.digi.xbee.api.models.XBee64BitAddress;

[...]

// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600);
myXBeeDevice.open();

// Get the XBee Network object from the XBee device.
XBeeNetwork network = myXBeeDevice.getNetwork();

// Discover devices in the network.
[...]
```

```
System.out.println("There are " + network.getNumberOfDevices() + " device(s) in
the network.");

// Get the remote XBee device whose 64-bit address is 0123456789ABCDEF.
RemoteXBeeDevice remoteDevice = network.getDevice(new XBee64BitAddress
("0123456789ABCDEF"));

[...]
```

Add and remove devices manually

This section provides information on methods for adding, removing and clearing the list of remote XBee devices.

Adding devices to the XBee network manually

There are several methods for adding remote XBee devices to an XBee network, in addition to the discovery methods provided by the XBeeNetwork object:

Method	Description
addRemoteDevice (RemoteXBeeDevice)	Specifies the remote XBee device to be added to the list of remote devices of the XBeeNetwork object. **Note** This operation does not join the remote XBee device to the network; it tells the network that it contains the device. However, the device has only been added to the device list and may not be physically in the same network.
addRemoteDevices (List<RemoteDevice>)	Specifies the list of remote XBee devices to be added to the list of remote devices of the XBeeNetwork object. **Note** This operation does not join the remote XBee devices to the network; it tells the network that it contains those devices. However, the devices have only been added to the device list and may not be physically in the same network.

```
import com.digi.xbee.api.RemoteXBeeDevice;
import com.digi.xbee.api.XBeeDevice;
import com.digi.xbee.api.XBeeNetwork;
import com.digi.xbee.api.models.XBee64BitAddress;

[...]

// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600);
myXBeeDevice.open();

// Get the XBee Network object from the XBee device.
XBeeNetwork network = myXBeeDevice.getNetwork();

// Instantiate a remote XBee device.
XBee64BitAddress remoteAddress = new XBee64BitAddress("0123456789ABCDEF");
```

```
RemoteXBeeDevice remoteDevice = new RemoteXBeeDevice(myXBeeDevice,
remoteAddress);

// Add the remote XBee device to the network.
network.addRemoteDevice(remoteDevice);

[...]
```

Removing an existing device from the XBee network

It is also possible to remove a remote XBee device from the list of remote XBee devices of the XBeeNetwork object by calling the following method:

Method	Description
removeRemoteDevice (RemoteXBeeDevice)	Specifies the remote XBee device to be removed from the list of remote devices of the XBeeNetwork object. If the device was not contained in the list the method will do nothing. **Note** This operation does not remove the remote XBee device from the actual XBee network; it tells the network object that it will no longer contain that device. However, next time you perform a discovery, it could be added again automatically.

```
import com.digi.xbee.api.RemoteXBeeDevice;
import com.digi.xbee.api.XBeeDevice;
import com.digi.xbee.api.XBeeNetwork;
import com.digi.xbee.api.models.XBee64BitAddress;

[...]

// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600);
myXBeeDevice.open();

// Get the XBee Network object from the XBee device.
XBeeNetwork network = myXBeeDevice.getNetwork();

// Discover devices in the network.
[...]

// Get the remote XBee device whose 64-bit address is 0123456789ABCDEF.
XBee64BitAddress remoteAddress = new XBee64BitAddress("0123456789ABCDEF");
RemoteXBeeDevice remoteDevice = network.getDeviceBy64BitAddress(remoteAddress);

// Remove the remote device from the network.
network.removeRemoteDevice(remoteDevice);

[...]
```

Clearing the list of remote XBee devices from the XBee network

The XBeeNetwork object also includes a method to clear the list of remote devices. This can be useful when you want to perform a clean discovery, cleaning the list before calling the discovery method.

Method	Description
clearDeviceList()	Removes all the devices from the list of remote devices of the network. **Note** This does not imply removing the XBee devices from the actual XBee network; it tells the object that the list should be empty now. Next time you perform a discovery, the list could be filled with the remote XBee devices found.

```
import com.digi.xbee.api.XBeeDevice;
import com.digi.xbee.api.XBeeNetwork;

[...]

// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600);
myXBeeDevice.open();

// Get the XBee Network object from the XBee device.
XBeeNetwork network = myXBeeDevice.getNetwork();

// Discover devices in the network.
[...]

// Clear the list of devices.
network.clearDeviceList();

[...]
```

Communicating with XBee devices

The XBee Java Library provides the ability to communicate with remote nodes in the network. The communication between XBee devices in a network involves the transmission and reception of data.

Communication features described in this topic and sub-topics are only applicable for local XBee devices. Remote XBee device classes do not include methods for transmitting or receiving data.

In this section, you will learn how to:

- Send data
- Send explicit data
- Send IP data
- Send SMS messages
- Receive data
- Receive explicit data
- Receive IP data

- Receive SMS messages
- Receive modem status events

Send data

A data transmission operation sends data from your local (attached) XBee device to a remote device on the network. The operation sends data in API frames, but the XBee Java library abstracts the process so your only concern is the node you want to send data to and the data itself.

You can send data either using a unicast or broadcast transmission. Unicast transmissions route data from one source device to one destination device, whereas broadcast transmissions are sent to all devices in the network.

Send data to one device

Unicast transmissions are sent from one source device to another destination device. The destination device could be an immediate neighbor of the source, or it could be several hops away.

Data transmission can be synchronous or asynchronous, depending on the method used.

Synchronous operation

This kind of operation is blocking. This means the method waits until the transmit status response is received or the default timeout is reached.

The XBeeDevice class of the API provides the following method to perform a synchronous unicast transmission with a remote node of the network:

Method	Description
sendData(RemoteXBeeDevice, byte[])	Specifies the remote XBee destination object and the data.

Protocol-specific classes offer additional synchronous unicast transmission methods apart from the one provided by the XBeeDevice object:

XBee class	Method	Description
ZigbeeDevice	**sendData (XBee64BitAddress, XBee16BitAddress, byte[])**	Specifies the 64-bit and 16-bit destination addresses and the data to send. If you do not know the 16-bit address, use the **XBee16BitAddress.UNKNOWN_ADDRESS**.
Raw802Device	**sendData (XBee16BitAddress, byte[]**)	Specifies the 16-bit destination address and the data to send.
	sendData (XBee64BitAddress, byte[])	Specifies the 64-bit destination address and the data to send.
DigiMeshDevice	**sendData (XBee64BitAddress, byte[])**	Specifies the 64-bit destination address and the data to send.

XBee class	Method	Description
DigiPointDevice	**sendData (XBee64BitAddress, XBee16BitAddress, byte[])**	Specifies the 64-bit and 16-bit destination addresses and the data to send. If the 16-bit address is unknown the **XBee16BitAddress.UNKNOWN_ADDRESS** can be used.

Sending data synchronously

```
import com.digi.xbee.api.RemoteXBeeDevice;
import com.digi.xbee.api.XBeeDevice;
import com.digi.xbee.api.models.XBee64BitAddress;

[...]

String data = "Hello XBee!";

// Instantiate an XBee device object.
XBeeDevice myLocalXBeeDevice = new XBeeDevice("COM1", 9600);
myLocalXBeeDevice.open();

// Instantiate a remote XBee device object.
RemoteXBeeDevice myRemoteXBeeDevice = new RemoteXBeeDevice(myLocalXBeeDevice,
                                   new XBee64BitAddress("0013A20040XXXXXX"));

// Send data using the remote object.
myLocalXBeeDevice.sendData(myRemoteXBeeDevice, data.getBytes());

[...]
```

The **sendData()** method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

The default timeout to wait for the send status is two seconds. However, you can configure the timeout using the **getReceivedTimeout** and **setReceiveTimeout** methods of an XBee device class.

Get/set the timeout for synchronous operations

```
import com.digi.xbee.api.XBeeDevice;

[...]

public static final int NEW_TIMEOUT_FOR_SYNC_OPERATIONS = 5 * 1000; // 5 seconds

XBeeDevice myXBeeDevice = [...]

// Retrieving the configured timeout for synchronous operations.
System.out.println("Current timeout: " + myXBeeDevice.getReceiveTimeout() + "
milliseconds.");

[...]

// Configuring the new timeout (in milliseconds) for synchronous operations.
myXBeeDevice.setReceiveTimeout(NEW_TIMEOUT_FOR_SYNC_OPERATIONS);

[...]
```

Synchronous Unicast Transmission Example

The XBee Java Library includes a sample application that shows you how to send data to another XBee device on the network. The example is located in the following path:

/examples/communication/SendDataSample

Asynchronous operation

Transmitting data asynchronously means that your application does not block during the transmit process. However, you cannot ensure that the data was successfully sent to the remote device.

The XBeeDevice class of the API provides the following method to perform an asynchronous unicast transmission with a remote node on the network:

Method	Description
sendDataAsync(RemoteXBeeDevice, byte[])	Specifies the remote XBee destination object and the data.

Protocol-specific classes offer some other asynchronous unicast transmission methods in addition to the one provided by the XBeeDevice object:

XBee class	Method	Description
ZigbeeDevice	**sendDataAsync (XBee64BitAddress, XBee16BitAddress, byte[])**	Specifies the 64-bit and 16-bit destination addresses and the data to send. If you do not know the 16-bit address, you can use **XBee16BitAddress.UNKNOWN_ADDRESS**.
Raw802Device	**sendDataAsync (XBee16BitAddress, byte[])**	Specifies the 16-bit destination address and the data to send.
	sendDataAsync (XBee64BitAddress, byte[])	Specifies the 64-bit destination address and the data to send.
DigiMeshDevice	**sendDataAsync (XBee64BitAddress, byte[])**	Specifies the 64-bit destination address and the data to send.
DigiPointDevice	**sendDataAsync (XBee64BitAddress, XBee16BitAddress, byte[])**	Specifies the 64-bit and 16-bit destination addresses and the data to send. If you do not know the 16-bit address, you can use **XBee16BitAddress.UNKNOWN_ADDRESS**.

Sending data asynchronously

```
import com.digi.xbee.api.RemoteXBeeDevice;
import com.digi.xbee.api.XBeeDevice;
import com.digi.xbee.api.models.XBee64BitAddress;

[...]

String data = "Hello XBee!";

// Instantiate an XBee device object.
XBeeDevice myLocalXBeeDevice = new XBeeDevice("COM1", 9600);
myLocalXBeeDevice.open();

// Instantiate a remote XBee device object.
RemoteXBeeDevice myRemoteXBeeDevice = new RemoteXBeeDevice(myLocalXBeeDevice,
                                   new XBee64BitAddress("000000409D5EXXXX"));

// Send data using the remote object.
myLocalXBeeDevice.sendDataAsync(myRemoteXBeeDevice, data.getBytes());
```

The **sendDataAsync()** method may fail for the following reasons:

- All the possible errors are caught as an **XBeeException**:
  - The operating mode of the device is not API or API_ESCAPE, throwing an **InvalidOperatingModeException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

Asynchronous Unicast Transmission Example

The XBee Java Library includes a sample application that shows you how to send data to another XBee device asynchronously. The example is located in the following path:

/examples/communication/SendDataAsyncSample

Send data to all devices of the network

Broadcast transmissions are sent from one source device to all the other devices on the network.

All the XBee device classes (generic and protocol specific) provide the same method to send broadcast data:

Method	Description
sendBroacastData(byte[])	Specifies the data to send.

Sending broadcast data

```
import com.digi.xbee.api.XBeeDevice;

[...]

String data = "Hello XBees!";

// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600);
myXBeeDevice.open();
```

```
// Send broadcast data.
myXBeeDevice.sendBroadcastData(data.getBytes());

[...]
```

The **sendBroadcastData()** method may fail for the following reasons:

- Transmit status is not received in the configured timeout, throwing a **TimeoutException** exception.
- Error types catch as **XBeeException**:
  - The operating mode of the device is not API or API_ESCAPE, throwing an **InvalidOperatingModeException**.
  - The transmit status is not **SUCCESS**, throwing a **TransmitException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

Broadcast Transmission Example

The XBee Java Library includes a sample application that shows you how to send data to all the devices on the network (broadcast). The example is located in the following path:

/examples/communication/SendBroadcastDataSample

Send explicit data

Some Zigbee applications may require communication with third-party (non-Digi) RF modules. These applications often send data of different public profiles such as Home Automation or Smart Energy to other modules.

XBee Zigbee modules offer a special type of frame for this purpose. Explicit frames transmit explicit data. When sending public profile packets, the frames transmit the data itself plus the application layer-specific fields—the source and destination endpoints, profile ID, and cluster ID.

Only Zigbee, DigiMesh, and Point-to-Multipoint protocols support the transmission of data in explicit format. This means you cannot transmit explicit data using a generic XBeeDevice object. You must use a protocol-specific XBee device object such as a ZigbeeDevice.

You can send explicit data as either unicast or broadcast transmissions. Unicast transmissions route data from one source device to one destination device, whereas broadcast transmissions are sent to all devices in the network.

Send explicit data to one device

Unicast transmissions are sent from one source device to another destination device. The destination device could be an immediate neighbor of the source, or it could be several hops away.

Unicast explicit data transmission can be a synchronous or asynchronous operation, depending on the method used.

Synchronous operation

The synchronous data transmission is a blocking operation. That is, the method waits until it either receives the transmit status response or the default timeout is reached.

All local XBee device classes that support explicit data transmission provide a method to transmit unicast and synchronous explicit data to a remote node of the network:

Method	Description
sendExplicitData (RemoteXBeeDevice, int, int, int, int, byte[])	Specifies remote XBee destination object, four application layer fields (source endpoint, destination endpoint, cluster ID, and profile ID), and data to send.

Every protocol-specific XBee device object with support for explicit data includes at least one more method to transmit unicast explicit data synchronously:

XBee class	Method	Description
ZigbeeDevice	**sendExplicitData (XBee64BitAddress, XBee16BitAddress, int, int, int, int, byte[])**	Specifies the 64-bit and 16-bit destination addresses in addition to the four application layer fields (source endpoint, destination endpoint, cluster ID, and profile ID) and the data to send. If the 16-bit address is unknown, use the **XBee16BitAddress.UNKNOWN_ADDRESS**.
DigiMeshDevice	**sendExplicitData (XBee64BitAddress, int, int, int, int, byte[])**	Specifies the 64-bit destination address, the four application layer fields (source endpoint, destination endpoint, cluster ID, and profile ID) and the data to send.
DigiPointDevice	**sendExplicitData (XBee64BitAddress, XBee16BitAddress, int, int, int, int, byte[])**	Specifies the 64-bit and 16-bit destination addresses in addition to the four application layer fields (source endpoint, destination endpoint, cluster ID, and profile ID) and the data to send. If the 16-bit address is unknown, use the **XBee16BitAddress.UNKNOWN_ADDRESS**.

Send unicast explicit data synchronously

```
import com.digi.xbee.api.RemoteZigbeeDevice;
import com.digi.xbee.api.ZigbeeDevice;
import com.digi.xbee.api.models.XBee64BitAddress;

[...]

String data = "Hello XBee!";

// Instantiate a Zigbee device object.
ZigbeeDevice myLocalZigbeeDevice = new ZigbeeDevice("COM1", 9600);
myLocalXBeeDevice.open();

// Instantiate a remote Zigbee device object.
RemoteXBeeDevice myRemoteXBeeDevice = new RemoteZigbeeDevice(myLocalXBeeDevice,
                                   new XBee64BitAddress("0013A20040XXXXXX"));

// Send explicit data synchronously using the remote object.
int sourceEndpoint = 0xA0;
int destinationEndpoint = 0xA1;
int clusterID = 0x1554;
int profileID = 0xC105;

myLocalZigbeeDevice.sendExplicitData(myRemoteXBeeDevice, sourceEndpoint,
                       destinationEndpoint, clusterID, profileID, data.getBytes());
```

```
[...]
```

The **sendExplicitData** method may fail for the following reasons:

- The method throws a **TimeoutException** exception if the response is not received in the configured timeout.
- Other errors register as **XBeeException**:
    - If the operating mode of the device is not API or API_ESCAPE, the method throws an **InvalidOperatingModeException**.
    - If the transmit status is not **SUCCESS**, the method throws a **TransmitException**.
    - If there is an error writing to the XBee interface, the method throws a generic **XBeeException**.

The default timeout to wait for send status is two seconds. You can configure this value using the **getReceivedTimeout** and **setReceiveTimeout** methods of a local XBee device class.

Get/set the timeout for synchronous operations

```
import com.digi.xbee.api.XBeeDevice;
[...]

public static final int NEW_TIMEOUT_FOR_SYNC_OPERATIONS = 5 * 1000; // 5 seconds

XBeeDevice myXBeeDevice = [...]

// Retrieving the configured timeout for synchronous operations.
System.out.println("Current timeout: " + myXBeeDevice.getReceiveTimeout() + "
milliseconds.");

[...]

// Configuring the new timeout (in milliseconds) for synchronous operations.
myXBeeDevice.setReceiveTimeout(NEW_TIMEOUT_FOR_SYNC_OPERATIONS);

[...]
```

Example: Transmit explicit synchronous unicast data

The XBee Java Library includes a sample application that demonstrates how to send explicit data to a remote device of the network (unicast). It can be located in the following path:

/examples/communication/explicit/SendExplicitDataSample

Asynchronous operation

Transmitting explicit data asynchronously means that your application does not block during the transmit process. However, you cannot ensure that the data was successfully sent to the remote device.

All local XBee device classes that support explicit data transmission provide a method to transmit unicast and asynchronous explicit data to a remote node of the network:

Method	Description
sendExplicitDataAsync (RemoteXBeeDevice, int, int, int, int, byte[])	Specifies the remote XBee destination object, four application layer fields (source endpoint, destination endpoint, cluster ID, and profile ID), and data to send.

Every protocol-specific XBee device object that supports explicit data includes at least one additional method to transmit unicast explicit data asynchronously:

XBee class	Method	Description
ZigbeeDevice	**sendExplicitDataAsync (XBee64BitAddress, XBee16BitAddress, int, int, int, int, byte [])**	Specifies the 64-bit and 16-bit destination addresses in addition to the four application layer fields (source endpoint, destination endpoint, cluster ID, and profile ID) and the data to send. If the 16-bit address is unknown, use the **XBee16BitAddress.UNKNOWN_ADDRESS**.
DigiMeshDevice	**sendExplicitDataAsync (XBee64BitAddress, int, int, int, int, byte [])**	Specifies the 64-bit destination address, the four application layer fields (source endpoint, destination endpoint, cluster ID, and profile ID) and the data to send.
DigiPointDevice	**sendExplicitDataAsync (XBee64BitAddress, XBee16BitAddress, int, int, int, int, byte [])**	Specifies the 64-bit and 16-bit destination addresses in addition to the four application layer fields (source endpoint, destination endpoint, cluster ID, and profile ID) and the data to send. If the 16-bit address is unknown, use the **XBee16BitAddress.UNKNOWN_ADDRESS**.

Send unicast explicit data asynchronously

```
import com.digi.xbee.api.RemoteZigbeeDevice;
import com.digi.xbee.api.ZigbeeDevice;
import com.digi.xbee.api.models.XBee64BitAddress;

[...]

String data = "Hello XBee!";

// Instantiate a Zigbee device object.
ZigbeeDevice myLocalZigbeeDevice = new ZigbeeDevice("COM1", 9600);
myLocalXBeeDevice.open();

// Instantiate a remote Zigbee device object.
RemoteXBeeDevice myRemoteXBeeDevice = new RemoteZigbeeDevice(myLocalXBeeDevice,
                                   new XBee64BitAddress("0013A20040XXXXXX"));

// Send explicit data asynchronously using the remote object.
int sourceEndpoint = 0xA0;
int destinationEndpoint = 0xA1;
int clusterID = 0x1554;
int profileID = 0xC105;

myLocalZigbeeDevice.sendExplicitDataAsync(myRemoteXBeeDevice, sourceEndpoint,
                     destinationEndpoint, clusterID, profileID, data.getBytes());
```

```
[...]
```

The **sendExplicitDataAsync** method may fail for the following reasons:

- All possible errors are caught as an **XBeeException**:
  - If the operating mode of the device is not API or API_ESCAPE, the method throws an **InvalidOperatingModeException**.
  - If there is an error writing to the XBee interface, the method throws a generic **XBeeException**.

Example: Transmit explicit asynchronous unicast data

The XBee Java API includes a sample application that demonstrates how to send explicit data to other XBee devices asynchronously. It can be located in the following path:

/examples/communication/explicit/SendExplicitDataAsyncSample

Send explicit data to all devices in the network

Broadcast transmissions are sent from one source device to all other devices in the network.

All protocol-specific XBee device classes that support the transmission of explicit data provide the same method to send broadcast explicit data:

Method	Description
sendBroacastExplicitData (int, int, int, int, byte[])	Specifies the four application layer fields (source endpoint, destination endpoint, cluster ID, and profile ID) and the data to send.

Send explicit broadcast data

```
import com.digi.xbee.api.ZigbeeDevice;

[...]

String data = "Hello XBees!";

// Instantiate a Zigbee device object.
ZigbeeDevice myXBeeDevice = new ZigbeeDevice("COM1", 9600);
myXBeeDevice.open();

// Send broadcast data.
int sourceEndpoint = 0xA0;
int destinationEndpoint = 0xA1;
int clusterID = 0x1554;
int profileID = 0xC105;

myXBeeDevice.sendBroadcastExplicitData(sourceEndpoint, destinationEndpoint
                         clusterID, profileID, data.getBytes());

[...]
```

The **sendBroadcastExplicitData** method may fail for the following reasons:

- If the transmit status is not received in the configured timeout, the method throws a **TimeoutException**.

- Other errors register as **XBeeException**:
  - If the operating mode of the device is not API or API_ESCAPE, the method throws an **InvalidOperatingModeException**.
  - If the transmit status is not **SUCCESS**, the method throws a **TransmitException**.
  - If there is an error writing to the XBee interface, the method throws a generic **XBeeException**.

Example: Send explicit broadcast data

The XBee Java Library includes a sample application that demonstrates how to send explicit data to all devices in the network (broadcast). It can be located in the following path:

/examples/communication/explicit/SendBroadcastExplicitDataSample

Send IP data

In contrast to XBee protocols like Zigbee, DigiMesh or 802.15.4, where the devices are connected each other, in Cellular and Wi-Fi protocols the modules are part of the Internet.

XBee Cellular and Wi-Fi modules offer a special type of frame for communicating with other Internet-connected devices. It allows sending data specifying the destination IP address, port, and protocol (TCP, TCP SSL or UDP).

Only Cellular, NB-IoT and Wi-Fi protocols support the transmission of IP data. This means you cannot transmit IP data using a generic XBeeDevice object; you must use the protocol-specific XBee device objects **CellularDevice**, **NBIoTDevice** or **WiFiDevice**.

IP data transmission can be a synchronous or asynchronous operation, depending on the method you use.

Synchronous Operation

The synchronous data transmission is a blocking operation; that is, the method waits until it either receives the transmit status response or it reaches the default timeout.

The **CellularDevice**, **NBIoTDevice** and **WiFiDevice** classes include several methods to transmit IP data synchronously:

Method	Description
sendIPData (Inet4Address, int, IPProtocol, byte[])	Specifies the destination IP address, destination port, IP protocol (UDP, TCP or TCP SSL) and data to send for transmissions.
sendIPData (Inet4Address, int, IPProtocol, boolean, byte[])	Specifies the destination IP address, destination port, IP protocol (UDP, TCP or TCP SSL), whether the socket should be closed after the transmission or not and data to send for transmissions.

Note NB-IoT modules only support UDP transmissions, so make sure that you use that protocol when calling the previous methods.

Send network data synchronously

```
import java.net.Inet4Address;

import com.digi.xbee.api.CellularDevice;
import com.digi.xbee.api.models.IPProtocol;

[...]

// Instantiate a Cellular device object.
CellularDevice myDevice = new CellularDevice("COM1", 9600);
myDevice.open();

// Send IP data using TCP.
Inet4Address destAddr = (Inet4Address) Inet4Address.getByName("56.23.102.96");
int destPort = 5050;
boolean closeSocket = false;
IPProtocol protocol = IPProtocol.TCP;
String data = "Hello XBee!";

myDevice.sendIPData(destAddr, destPort, closeSocket, protocol, data.getBytes());

[...]
```

The previous may fail for the following reasons:

There is a timeout setting the IP addressing parameter, throwing a **TimeoutException**.

- Other errors caught as XBeeException:
  - The operating mode of the device is not API or API_ESCAPE, throwing an **InvalidOperatingModeException**.
  - If the transmit status is not SUCCESS, the method throws a **TransmitException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

The default timeout to wait for the send status is two seconds. You can configure this value using the **getReceiveTimeout** and **setReceiveTimeout** methods of a local XBee device class.

Get/set the timeout for synchronous operations

```
import com.digi.xbee.api.CellularDevice;

[...]

public static final int NEW_TIMEOUT_FOR_SYNC_OPERATIONS = 5 * 1000; // 5 seconds

CellularDevice myDevice = [...]

// Retrieving the configured timeout for synchronous operations.
System.out.println("Current timeout: " + myDevice.getReceiveTimeout() + "
milliseconds.");

[...]

// Configuring the new timeout (in milliseconds) for synchronous operations.
myDevice.setReceiveTimeout(NEW_TIMEOUT_FOR_SYNC_OPERATIONS);

[...]
```

Example: Transmit IP data synchronously

The XBee Java Library includes a sample application that demonstrates how to send IP data. You can locate the example in the following path:

/examples/communication/ip/SendIPDataSample

Example: Transmit UDP data

The XBee Java Library includes a sample application that demonstrates how to send UDP data. You can locate the example in the following path:

/examples/communication/ip/SendUDPDataSample

Example: Connect to echo server

The XBee Java Library includes a sample application that demonstrates how to connect to an echo server, send a message to it and receive its response. You can locate the example in the following path:

/examples/communication/ip/ConnectToEchoServerSample

Example: Knock knock

The XBee Java Library includes a sample application that demonstrates how to connect to a web server and establish a conversation with knock-knock jokes. You can locate the example in the following path:

/examples/communication/ip/KnockKnockSample

Asynchronous operation

Transmitting IP data asynchronously means that your application does not block during the transmit process. However, you cannot ensure that the data was successfully sent.

The **CellularDevice**, **NBIoTDevice**, and **WiFiDevice** classes include several methods to transmit IP data asynchronously:

Method	Description
sendIPDataAsync (Inet4Address, int, IPProtocol, byte[])	Specifies the destination IP address, destination port, IP protocol (UDP, TCP or TCP SSL) and data to send for asynchronous transmissions.
sendIPDataAsync (Inet4Address, int, IPProtocol, boolean, byte[])	Specifies the destination IP address, destination port, IP protocol (UDP, TCP or TCP SSL), whether the socket should be closed after the transmission or not and data to send for asynchronous transmissions.

Note NB-IoT modules only support UDP transmissions, so make sure that you use that protocol when calling the previous methods.

Send network data asynchronously

```
import java.net.Inet4Address;
import com.digi.xbee.api.CellularDevice;

import com.digi.xbee.api.models.IPProtocol;
```

```
[...]

// Instantiate a Cellular device object.
CellularDevice myDevice = new CellularDevice("COM1", 9600);
myDevice.open();

// Send IP data using TCP.
Inet4Address destAddr = (Inet4Address) Inet4Address.getByName("56.23.102.96");
int destPort = 5050;
boolean closeSocket = false;
IPProtocol protocol = IPProtocol.TCP;
String data = "Hello XBee!";

myDevice.sendIPDataAsync(destAddr, destPort, protocol, closeSocket, data.getBytes
());

[...]
```

The previous methods may fail for the following reasons:

- All possible errors are caught as an **XBeeException**:
  - If the operating mode of the device is not **API** or **API_ESCAP**E, the method throws an **InvalidOperatingModeException**.
  - If there is an error writing to the XBee interface, the method throws a generic **XBeeException**.

Send IPv6 data

The XBee Thread radio modules use the IPv6 network protocol instead of IPv4 to communicate between modules. These modules allow sending data in a similar manner as IPv4 devices, but it is necessary to specify an IPv6 address.

Note Only Thread protocol supports the transmission of IPv6 data. This means you cannot transmit IPv6 data using a generic XBeeDevice object; you must use the protocol-specific XBee device object **ThreadDevice**.

IPv6 data transmission can be a synchronous or asynchronous operation, depending on the method you use.

Synchronous operation

The synchronous IPv6 data transmission is a blocking operation; that is, the method waits until it either receives the transmit status response or it reaches the default timeout.

The **ThreadDevice** class includes the following method to transmit IPv6 data synchronously:

Method	Description
sendIPData (Inet6Address, int, IPProtocol, byte[])	Specifies the destination IPv6 address, destination port, IP protocol (UDP, TCP, TCP SSL or CoAP) and data to send for transmissions.

Send IPv6 data synchronously

```
import java.net.Inet6Address;


import com.digi.xbee.api.ThreadDevice;
import com.digi.xbee.api.models.IPProtocol;

[...]

// Instantiate a Thread device object.
ThreadDevice myDevice = new ThreadDevice("COM1", 9600);
myDevice.open();

// Send IPv6 data using UDP.
Inet6Address destAddr = (Inet6Address) Inet6Address.getByName
("FDB3:0001:0002:0000:0004:0005:0006:0007");
int destPort = 9750;
IPProtocol protocol = IPProtocol.UDP;
String data = "Hello XBee!";

myDevice.sendIPData(destAddr, destPort, protocol, data.getBytes());

[...]
```

The previous method may fail for the following reasons:

- Transmit status of the packet sent is not received, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - If the transmit status is not **SUCCESS**, the method throws a **TransmitException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

The default timeout to wait for the send status is two seconds. You can configure this value using the **getReceiveTimeout** and **setReceiveTimeout** methods of a local XBee device class.

Get/set the timeout for synchronous operations

```
import com.digi.xbee.api.ThreadDevice;

[...]

public static final int NEW_TIMEOUT_FOR_SYNC_OPERATIONS = 5 * 1000; // 5 seconds

ThreadDevice myDevice = [...]

// Retrieving the configured timeout for synchronous operations.
System.out.println("Current timeout: " + myDevice.getReceiveTimeout() + "
milliseconds.");

[...]

// Configuring the new timeout (in milliseconds) for synchronous operations.
myDevice.setReceiveTimeout(NEW_TIMEOUT_FOR_SYNC_OPERATIONS);

[...]
```

Example: Transmit IPv6 data synchronously

The XBee Java Library includes a sample application that demonstrates how to send IPv6 data. You can locate the example in the following path:

/examples/communication/ip/SendIPv6DataSample

Asynchronous operation

Transmitting IPv6 data asynchronously means that your application does not block during the transmit process. However, you cannot ensure that the data was sent successfully.

The **ThreadDevice** class includes the following method to transmit IPv6 data asynchronously:

Method	Description
sendIPDataAsync (Inet6Address, int, IPProtocol, byte[])	Specifies the destination IPv6 address, destination port, IP protocol (UDP, TCP, TCP SSL or CoAP) and data to send for asynchronous transmissions.

Send IPv6 data asynchronously

```
import java.net.Inet6Address;

import com.digi.xbee.api.ThreadDevice;
import com.digi.xbee.api.models.IPProtocol;

[...]

// Instantiate a Thread device object.
ThreadDevice myDevice = new ThreadDevice("COM1", 9600);
myDevice.open();

// Send IPv6 data using UDP.
Inet6Address destAddr = (Inet6Address) Inet6Address.getByName
("FDB3:0001:0002:0000:0004:0005:0006:0007");
int destPort = 9750;
IPProtocol protocol = IPProtocol.UDP;
String data = "Hello XBee!";

myDevice.sendIPDataAsync(destAddr, destPort, protocol, data.getBytes());

[...]
```

The previous method may fail for the following reasons:

- All possible errors are caught as an **XBeeException**:
  - If the operating mode of the device is not **API** or **API_ESCAPE**, the method throws an **InvalidOperatingModeException**.
  - If there is an error writing to the XBee interface, the method throws a generic **XBeeException**.

Send CoAP data

Constrained Application Protocol (CoAP) is an application layer protocol used by devices with limited RAM and Flash capacity to interact with the Internet. It uses a client-server model where a client sends requests to a server, and the server sends back acknowledgments and responses.

Digi's XBee Thread radio modules support this protocol, and the XBee Java Library provides an API to send CoAP data.

Note Only the Thread protocol supports the transmission of CoAP data. This means you cannot transmit CoAP data using a generic XBeeDevice object; you must use the protocol-specific XBee device object ThreadDevice.

CoAP data transmission can be a synchronous or asynchronous operation, depending on the method you use.

Synchronous operation

The synchronous CoAP data transmission is a blocking operation; that is, the method waits until it either receives the transmit status response and the CoAP response or it reaches the default timeout.

The **ThreadDevice** class includes the following method to transmit CoAP data synchronously:

Method	Description
sendCoAPData (Inet6Address, String, HTTPMethod, byte[])	Specifies the destination IPv6 address, URI, HTTP method (EMPTY, GET, POST, PUT or DELETE) and payload to send.
sendCoAPData (Inet6Address, String, HTTPMethod, boolean, byte[])	Specifies the destination IPv6 address, URI, HTTP method (EMPTY, GET, POST, PUT or DELETE), whether to apply remote AT command changes and payload to send. This method should be used only when setting a remote AT command.

The Uniform Resource Identifier (URI) is a printable string that must be present in each CoAP transmission. The XBee Java Library provides some fixed URIs for the transmissions:

- **CoAPURI.URI_DATA_TRANSMISSION**: "XB/TX" for data transmissions (use PUT as HTTP method)
- **CoAPURI.URI_AT_COMMAND**: "XB/AT" for AT Command operation (use PUT to set an AT command or GET to read an AT command). After the URI, an AT command needs to be specified, for example:
  - CoAPURI.URI_AT_COMMAND + "/NI"
- **CoAPURI.URI_IO_SAMPLING**: "XB/IO" for IO operation (use POST as HTTP method)

Send CoAP data synchronously

```
import java.net.Inet6Address;

import com.digi.xbee.api.ThreadDevice;
import com.digi.xbee.api.models.HTTPMethod;

[...]

// Instantiate a Thread device object.
ThreadDevice myDevice = new ThreadDevice("COM1", 9600);
myDevice.open();
```

```
// Send IPv6 data using UDP. Set the remote AT command "NI" with "Device 1".
Inet6Address destAddr = (Inet6Address) Inet6Address.getByName
("FDB3:0001:0002:0000:0004:0005:0006:0007");
HTTPMethod method = HTTPMethod.PUT;
String uri = CoAPURI.URI_AT_COMMAND + "/NI";
String data = "Device 1";

myDevice.sendCoAPData(destAddr, uri, method, true, data.getBytes());

[...]
```

The previous method may fail for the following reasons:

- Transmit status of the packet sent is not received, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - If the transmit status is not **SUCCESS**, the method throws a **TransmitException**.
  - There is an error writing to the XBee interface or the CoAP response is not received, throwing a generic **XBeeException**.

The default timeout to wait for the send status is two seconds. You can configure this value using the **getReceiveTimeout** and **setReceiveTimeout** methods of a local XBee device class.

Get/set the timeout for synchronous operations

```
import com.digi.xbee.api.ThreadDevice;

[...]

public static final int NEW_TIMEOUT_FOR_SYNC_OPERATIONS = 5 * 1000; // 5 seconds

ThreadDevice myDevice = [...]

// Retrieving the configured timeout for synchronous operations.
System.out.println("Current timeout: " + myDevice.getReceiveTimeout() + "
milliseconds.");

[...]

// Configuring the new timeout (in milliseconds) for synchronous operations.
myDevice.setReceiveTimeout(NEW_TIMEOUT_FOR_SYNC_OPERATIONS);

[...]
```

Example: Transmit CoAP data synchronously

The XBee Java Library includes a sample application that demonstrates how to send CoAP data. You can locate the example in the following path:

/examples/communication/coap/SendCoAPDataSample

Asynchronous operation

Transmitting CoAP data asynchronously means that your application does not block during the transmit process. However, you cannot ensure that the data was successfully sent.

The ThreadDevice class includes the following method to transmit CoAP data asynchronously:

Method	Description
sendCoAPDataAsync (Inet6Address, String, HTTPMethod, byte [])	Specifies the destination IPv6 address, URI, HTTP method (EMPTY, GET, POST, PUT or DELETE) and payload to send for asynchronous transmissions.
sendCoAPDataAsync (Inet6Address, String, HTTPMethod, boolean, byte[])	Specifies the destination IPv6 address, URI, HTTP method (EMPTY, GET, POST, PUT or DELETE), whether to apply remote AT command changes and payload to send for asynchronous transmissions. This method should be used only when setting a remote AT command.

The Uniform Resource Identifier (URI) is a printable string that must be present in each CoAP transmission. The XBee Java Library provides some fixed URIs for the transmissions:

- **CoAPURI.URI_DATA_TRANSMISSION**: "XB/TX" for data transmissions (use PUT as HTTP method)
- **CoAPURI.URI_AT_COMMAND**: "XB/AT" for AT Command operation (use PUT to set an AT command or GET to read an AT command). After the URI, an AT command needs to be specified, for example:
  - CoAPURI.URI_AT_COMMAND + "/NI"
- **CoAPURI.URI_IO_SAMPLING**: "XB/IO" for IO operation (use POST as HTTP method)

Send CoAP data asynchronously

```
import java.net.Inet6Address;

import com.digi.xbee.api.ThreadDevice;
import com.digi.xbee.api.models.HTTPMethod;

[...]

// Instantiate a Thread device object.
ThreadDevice myDevice = new ThreadDevice("COM1", 9600);
myDevice.open();


// Send IPv6 data using UDP. Set the remote AT command "NI" with "Device 1".
Inet6Address destAddr = (Inet6Address) Inet6Address.getByName
("FDB3:0001:0002:0000:0004:0005:0006:0007");
HTTPMethod method = HTTPMethod.PUT;
String uri = CoAPURI.URI_AT_COMMAND + "/NI";
String data = "Device 1";

myDevice.sendCoAPDataAsync(destAddr, uri, method, true, data.getBytes());

[...]
```

The previous method may fail for the following reasons:

- All possible errors are caught as an **XBeeException**:
  - If the operating mode of the device is not **API** or **API_ESCAPE**, the method throws an **InvalidOperatingModeException**.

- If there is an error writing to the XBee interface, the method throws a generic **XBeeException**.

Send SMS messages

Another feature of the XBee Cellular module is the ability to send and receive Short Message Service (SMS) transmissions. This allows you to send and receive text messages to and from an SMS capable device such as a mobile phone.

For that purpose, these modules offer a special type of frame for sending text messages, specifying the destination phone number and data.

CAUTION! Only Cellular protocol supports the transmission of SMS. This means you cannot send text messages using a generic **XBeeDevice** object; you must use the protocol-specific XBee device object **CellularDevice**.

SMS transmissions can be a synchronous or asynchronous operation, depending on the used method.

Synchronous Operation

The synchronous SMS transmission is a blocking operation; that is, the method waits until it either receives the transmit status response or it reaches the default timeout.

The **CellularDevice** class includes the following method to send SMS messages synchronously:

Method	Description
sendSMS(String, String)	Specifies the the phone number to send the SMS to and the data to send as the body of the SMS message.

Send SMS message synchronously

```
import com.digi.xbee.api.CellularDevice;

[...]

// Instantiate a Cellular device object.
CellularDevice myDevice = new CellularDevice("COM1", 9600);
myDevice.open();

String phoneNumber = "+34665963205";
String data = "Hello XBee!";

// Send SMS message.
myDevice.sendSMS(phoneNumber, data);

[...]
```

The **sendSMS** method may fail for the following reasons:

- If the response is not received in the configured timeout, the method throws a **TimeoutException** exception.
- If the phone number has an invalid format, the method throws an **IllegalArgumentException** exception.

- Errors register as **XBeeException**:
  - If the operating mode of the device is not **API** or **API_ESCAPE**, the method throws an **InvalidOperatingModeException**.
  - If the transmit status is not SUCCESS, the method throws a **TransmitException**.
  - If there is an error writing to the XBee interface, the method throws a generic **XBeeException**.

The default timeout to wait for send status is two seconds. You can configure this value using the **getReceivedTimeout** and **setReceiveTimeout** methods of a local XBee device class.

Get/set the timeout for synchronous operations

```
import com.digi.xbee.api.CellularDevice;

[...]

public static final int NEW_TIMEOUT_FOR_SYNC_OPERATIONS = 5 * 1000; // 5 seconds

CellularDevice myDevice = [...]

// Retrieving the configured timeout for synchronous operations.
System.out.println("Current timeout: " + myDevice.getReceiveTimeout() + "
milliseconds.");

[...]

// Configuring the new timeout (in milliseconds) for synchronous operations.
myDevice.setReceiveTimeout(NEW_TIMEOUT_FOR_SYNC_OPERATIONS);

[...]
```

Example: Send synchronous SMS

The XBee Java Library includes a sample application that demonstrates how to send SMS messages. You can locate the example in the following path:

/examples/communication/cellular/SendSMSSample

Asynchronous operation

Transmitting SMS messages asynchronously means that your application does not block during the transmit process. However, you cannot verify the SMS was successfully sent.

The **CellularDevice** class includes the following method to send SMS asynchronously:

Method	Description
sendSMSAsync (String, String)	Specifies the the phone number to send the SMS to and the data to send as the body of the SMS message.

Send SMS message asynchronously

```
import com.digi.xbee.api.CellularDevice;

[...]
```

```
// Instantiate a Cellular device object.
CellularDevice myDevice = new CellularDevice("COM1", 9600);
myDevice.open();

String phoneNumber = "+34665963205";
String data = "Hello XBee!";

// Send SMS message.
myDevice.sendSMSAsync(phoneNumber, data);

[...]
```

The previous method may fail for the following reasons:

- If the phone number has an invalid format, the method throws an **IllegalArgumentException** exception.
- Errors register as **XBeeException**:
  - If the operating mode of the device is not **API** or **API_ESCAPE**, the method throws an **InvalidOperatingModeException**.
  - If there is an error writing to the XBee interface, the method throws a generic **XBeeException**.

Receive data

The data reception operation allows you to receive and handle data sent by other remote nodes of the network.

There are two different ways to read data from the device:

- Polling for data. This mechanism allows you to read (ask) for new data in a polling sequence. The read method blocks until data is received or until a configurable timeout has expired.
- Data reception callback. In this case, you must register a listener that executes a callback each time new data is received by the local XBee device (that is, the device attached to your PC) providing data and other related information.

Polling for data

The simplest way to read for data is by executing the **readData** method of the local XBee device. This method blocks your application until data from any XBee device of the network is received or the timeout provided has expired:

Method	Description
readData (int)	Specifies the time to wait for data reception (method blocks during that time or until data is received). If you do not specify a timeout the method uses the default receive timeout configured in the **XBeeDevice**.

Reading data from any remote XBee device (polling)

```
import com.digi.xbee.api.XBeeDevice;
import com.digi.xbee.api.models.XBeeMessage;
```

```
[...]

// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600);
myXBeeDevice.open();

// Read data.
XBeeMessage xbeeMessage = myXBeeDevice.readData();

[...]
```

The method returns the read data inside an **XBeeMessage** object. This object contains the following information:

- **RemoteXBeeDevice** that sent the message.
- Byte array with the contents of the received data.
- Flag indicating if the data was sent via broadcast.

You can retrieve the previous information using the corresponding getters of the **XBeeMessage** object:

Get the XBeeMessage information

```
import com.digi.xbee.api.XBeeAddress;
import com.digi.xbee.api.XBeeDevice;
import com.digi.xbee.api.models.XBeeMessage;

[...]

// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = [...]

// Read data.
XBeeMessage xbeeMessage = myXBeeDevice.readData();

RemoteXBeeDevice remote = xbeeMessage.getDevice();
byte[] data = xbeeMessage.getData();
boolean isBroadcast = xbeeMessage.isBroadcast();

[...]
```

You can also read data from a specific remote XBee device of the network. For that purpose, the XBee device object provides the **readDataFrom method**:

Method	Description
readDataFrom (RemoteXBeeDevice, int)	Specifies the remote XBee device to read data from and the time to wait for data reception (method blocks during that time or until data is received). If you do not specify a timeout the method uses the default receive timeout configured in the **XBeeDevice**.

Read data from a specific remote XBee device (polling)

```
import com.digi.xbee.api.RemoteXBeeDevice;
import com.digi.xbee.api.XBeeDevice;
import com.digi.xbee.api.models.XBeeMessage;
```

```
[...]

// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600);
myXBeeDevice.open();

// Instantiate a remote XBee device object.
RemoteXBeeDevice myRemoteXBeeDevice = [...]

// Read data sent by the remote XBee device.
XBeeMessage xbeeMessage = myXBeeDevice.readDataFrom(myRemoteXBeeDevice);

[...]
```

As in the previous method, this method also returns an **XBeeMessage** object with all the information inside.

In either case, the default timeout to wait for data is two seconds. However, it can be consulted and configured using the **getReceiveTimeout** and **setReceiveTimeout** methods of an XBee device class.

Get/set the timeout for synchronous operations

```
import com.digi.xbee.api.XBeeDevice;

[...]

public static final int NEW_TIMEOUT_FOR_SYNC_OPERATIONS = 5 * 1000; // 5 seconds

XBeeDevice myXBeeDevice = [...]

// Retrieving the configured timeout for synchronous operations.
System.out.println("Current timeout: " + myXBeeDevice.getReceiveTimeout() + "
milliseconds.");

[...]

// Configuring the new timeout (in milliseconds) for synchronous operations
myXBeeDevice.setReceiveTimeout(NEW_TIMEOUT_FOR_SYNC_OPERATIONS);

[...]
```

Data reception polling example

The XBee Java Library includes a sample application that shows you how to receive data using the polling mechanism. The example is located in the following path:

/examples/communication/ReceiveDataPollingSample

Data reception callback

This second mechanism to read data does not block your application. Instead, you can be notified when new data has been received if you are subscribed or registered to the data reception service using the **addDataListener(IDataReceiveListener)** method with a data reception listener as parameter.

Data reception registration

```
import com.digi.xbee.api.XBeeDevice;
[...]

// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600);
myXBeeDevice.open();

// Create the data reception listener.
MyDataReceiveListener myDataReceiveListener = ...

// Subscribe to data reception.
myXBeeDevice.addDataListener(myDataReceiveListener);

[...]
```

The listener that is provided to the subscribe method, **MyDataReceiveListener**, must implement the **IDataReceiveListener** interface. This interface includes the method executed when new data is received by the XBee device.

It does not matter which type of local XBee device you have instanced, as this data reception operation is implemented in the same way for all the local XBee device classes that support the receive data mechanism.

When new data is received, the **dataReceived()** method of the **IDataReceiveListener** is executed providing an **XBeeMessage** object as a parameter, which contains the data and other useful information.

IDataReceiveListener implementation example, MyDataReceiveListener

```
import com.digi.xbee.api.listeners.IDataReceiveListener;

public class MyDataReceiveListener implements IDataReceiveListener {
       /*
       * Data reception callback.
       */
       @Override
       public void dataReceived(XBeeMessage xbeeMessage) {
               String address = xbeeMessage.getDevice().get64BitAddress().toString();
               String dataString = xbeeMessage.getDataString();
               System.out.println("Received data from " + address +
                               ": " + dataString);
       }
}
```

The **XBeeMessage** object provides the following information:

- **RemoteXBeeDevice** that sent the message.
- Byte array with the contents of the received data.
- Flag indicating if the data was sent via broadcast.

You can retrieve the previous information using the corresponding getters of the **XBeeMessage** object:

Get the XBeeMessage information

```
[...]
```

```
XBeeDevice myXBeeDevice = ...
MyDataReceiveListener myDataReceiveListener = ...

myXBeeDevice.addDataListener(myDataReceiveListener);

[...]

// Remove the new data reception listener

myXBeeDevice.removeDataListener(myDataReceiveListener);

[...]
```

To stop listening to new received data, use the **removeDataListener(IDataReceiveListener)** method to unsubscribe the already registered listener.

Data reception deregistration

```
[...]

XBeeDevice myXBeeDevice = ...
MyDataReceiveListener myDataReceiveListener = ...

myXBeeDevice.addDataListener(myDataReceiveListener);

[...]

// Remove the new data reception listener.
myXBeeDevice.removeDataListener(myDataReceiveListener);

[...]
```

Data reception callback example

The XBee Java Library includes a sample application that shows you how to subscribe to the data reception service to receive data. The example is located in the following path:

/examples/communication/ReceiveDataSample

Receive explicit data

Some applications developed with the XBee Java Library may require modules to receive data in application layer, or explicit, data format.

Only Zigbee, DigiMesh, and Point-to-Multipoint support the reception of explicit data.

To receive data in explicit format, you must first configure the data output mode of the receiver XBee device to explicit format using the **setAPIOutputMode** method.

Method	Description
getAPIOutputMode ()	Returns the API output mode of the data received by the XBee device.
setAPIOutputMode (APIOutputMode)	Specifies the API output mode of the data received by the XBee device. The mode can be one of the following: ■ **APIOutputMode.NATIVE**: The data received by the device will be output as standard received data and it must be read using standard data-reading methods. It does not matter if the data sent by the remote device was sent in standard or explicit format. ■ **APIOutputMode.EXPLICIT**: The data received by the device will be output as explicit received data and it must be read using explicit data-reading methods. It does not matter if the data sent by the remote device was sent in standard or explicit format. ■ **APIOutputMode.EXPLICIT_ZDO_PASSTHRU**: The data received by the device will be output as explicit received data, like the **APIOutputMode.EXPLICIT** option. In addition, this mode also outputs as explicit data Zigbee Device Object (ZDO) packets received by the XBee module through the serial interface.

Once you have configured the device to receive data in explicit format, you can read it using one of the following mechanisms provided by the XBee device object:

Polling for data

The simplest way to read for explicit data is by executing the **readExplicitData** method of the local XBee device. This method blocks your application until explicit data from any XBee device of the network is received or the provided timeout has expired:

Method	Description
readExplicitData (int)	Specifies the time to wait in milliseconds for explicit data reception (method blocks during that time or until explicit data is received). If you don't specify a timeout, the method uses the default receive timeout configured in **XBeeDevice**.

Read explicit data from any remote XBee device (polling)

```
import com.digi.xbee.api.ZigbeeDevice;
import com.digi.xbee.api.models.ExplicitXBeeMessage;

[...]

// Instantiate a Zigbee device object.
ZigbeeDevice myZigbeeDevice = new ZigbeeDevice("COM1", 9600);
myZigbeeDevice.open();

// Read explicit data.
ExplicitXBeeMessage xbeeMessage = myZigbeeDevice.readExplicitData();
```

```
[...]
```

The read data is returned inside an **ExplicitXBeeMessage** object. This object also contains the application layer fields as well as the following information:

- RemoteXBeeDevice that sent the data.
- Endpoint of the source that initiated the transmission.
- Endpoint of the destination where the message is addressed.
- Cluster ID where the data was addressed.
- Profile ID where the data was addressed.
- Byte array with the contents of the received data.
- Flag indicating if the data was sent via broadcast.

You can retrieve the previous information using the corresponding getters of the **ExplicitXBeeMessage** object:

Get the ExplicitXBeeMessage information

```
import com.digi.xbee.api.ZigbeeDevice;
import com.digi.xbee.api.XBeeAddress;
import com.digi.xbee.api.models.ExplicitXBeeMessage;

[...]

// Instantiate a Zigbee device object.
ZigbeeDevice myZigbeeDevice = [...]

// Read explicit data.
ExplicitXBeeMessage xbeeMessage = myZigbeeDevice.readExplicitData();

RemoteXBeeDevice remote = xbeeMessage.getDevice();
int sourceEndpoint = xbeeMessage.getSourceEndpoint();
int destEndpoint = xbeeMessage.getDestinationEndpoint();
int clusterID = xbeeMessage.getClusterID();
int profileID = xbeeMessage.getProfileID();
byte[] data = xbeeMessage.getData();
boolean isBroadcast = xbeeMessage.isBroadcast();

[...]
```

You can also read explicit data from a specific remote XBee device of the network. For that purpose, the XBee device object provides the **readExplicitDataFrom** method:

Method	Description
readExplicitDataFrom (RemoteXBeeDevice, int)	Specifies the remote XBee device to read explicit data from and the time to wait for explicit data reception (method blocks during that time or until explicit data is received). If you do not specify a timeout, the method uses the default receive timeout configured in the XBee device object.

Read explicit data from a specific remote XBee device (polling)

```
import com.digi.xbee.api.RemoteXBeeDevice;
import com.digi.xbee.api.ZigbeeDevice;
import com.digi.xbee.api.models.ExplicitXBeeMessage;

[...]

// Instantiate a Zigbee device object.
ZigbeeDevice myZigbeeDevice = new ZigbeeDevice("COM1", 9600);
myZigbeeDevice.open();

// Instantiate a remote XBee device object.
RemoteXBeeDevice myRemoteXBeeDevice = [...]

// Read data sent by the remote XBee device.
ExplicitXBeeMessage xbeeMessage = myZigbeeDevice.readExplicitDataFrom
(myRemoteXBeeDevice);

[...]
```

This method also returns an **ExplicitXBeeMessage** object containing the same information as the **ExplicitXBeeMessage** object returned by the **readExplicitData** method.

In either case, the default timeout to wait for data is two seconds. You can configure this timeout with the **getReceiveTimeout** and **setReceiveTimeout** methods of an XBee device class.

Get/set the timeout for synchronous operations

```
import com.digi.xbee.api.ZigbeeDevice;

[...]

public static final int NEW_TIMEOUT_FOR_SYNC_OPERATIONS = 5 * 1000; // 5 seconds

ZigbeeDevice myZigbeeDevice = [...]

// Retrieving the configured timeout for synchronous operations.
System.out.println("Current timeout: " + myZigbeeDevice.getReceiveTimeout() + "
milliseconds.");

[...]

// Configuring the new timeout (in milliseconds) for synchronous operations.
myZigbeeDevice.setReceiveTimeout(NEW_TIMEOUT_FOR_SYNC_OPERATIONS);

[...]
```

Example: Receive explicit data with polling

The XBee Java Library includes a sample application that demonstrates how to receive explicit data using the polling mechanism. It can be located in the following path:

/examples/communication/explicit/ReceiveExplicitDataPollingSample

Explicit data reception callback

This mechanism for reading explicit data does not block your application. Instead, you can be notified when new explicit data has been received if you are subscribed or registered to the explicit data reception service by the **addExplicitDataListener(IExplicitDataReceiveListener)**.

Explicit data reception registration

```
import com.digi.xbee.api.ZigbeeDevice:
import com.digi.xbee.api.listeners.IExplicitDataReceiveListener;

[...]

// Instantiate a Zigbee device object.
ZigbeeDevice myZigbeeDevice = new ZigbeeDevice("COM1", 9600);
myZigbeeDevice.open();

// Subscribe to explicit data reception.
myZigbeeDevice.addExplicitDataListener(new MyExplicitDataReceiveListener());

[...]
```

The listener provided to the subscribed method, **MyExplicitDataReceiveListener**, must implement the **IExplicitDataReceiveListener** interface. This interface includes the method that is executed when new explicit data is received by the XBee device.

This explicit data reception operation is implemented the same way for all local XBee device classes.

Remember that 802.15.4, Cellular and Wi-Fi protocols do not support transmitting explicit data, so you cannot use the methods explained in this section when working with these protocols.

When new explicit data is received, the **explicitDataReceived()** method of the **IExplicitDataReceiveListener** is executed providing as parameter an **ExplicitXBeeMessage** object which contains the data and other useful information such as the application layer fields.

ExplicitDataReceiveListener implementation example

```
import com.digi.xbee.api.listeners.IExplicitDataReceiveListener;
import com.digi.xbee.api.models.ExplicitXBeeMessage;

public class MyExplicitDataReceiveListener implements
IExplicitDataReceiveListener {
       /*
       * Explicit data reception callback.
       */
       @Override
       public void explicitDataReceived(ExplicitXBeeMessage xbeeMessage) {
               String address = xbeeMessage.getDevice().get64BitAddress().toString();
               int sourceEndpoint = xbeeMessage.getSourceEndpoint();
               int destEndpoint = xbeeMessage.getDestinationEndpoint();
               int cluster = xbeeMessage.getClusterID();
               int profile = xbeeMessage.getProfileID();
               String dataString = xbeeMessage.getDataString();
               System.out.println("Received explicit data from " + address +
                                                               ": " + dataString);
               System.out.println("Application layer fields:");
               System.out.println(" - Source endpoint: " + HexUtils.integerToHexString
(sourceEndpoint, 1));
               System.out.println(" - Destination endpoint: " + HexUtils.integerToHexString
(destEndpoint, 1));
               System.out.println(" - Cluster ID: " + HexUtils.integerToHexString(cluster,
2));
```

```
                System.out.println(" - Profile ID: " + HexUtils.integerToHexString(profile,
2));
        }
}
```

The ExplicitXBeeMessage object provides the following information:

- RemoteXBeeDevice that sent the data
- Endpoint of the source that initiated the transmission
- Endpoint of the destination where the message is addressed
- Cluster ID where the data was addressed
- Profile ID where the data was addressed
- Byte array with the contents of the received data
- Flag indicating if the data was sent via broadcast

You can retrieve the previous information using the corresponding getters of the **ExplicitXBeeMessage** object:

Get the ExplicitXBeeMessage information

```
import com.digi.xbee.api.RemoteXBeeDevice;
import com.digi.xbee.api.listeners.IExplicitDataReceiveListener;
import com.digi.xbee.api.models.ExplicitXBeeMessage;

public class MyExplicitDataReceiveListener implements
IExplicitDataReceiveListener {
        /*
        * Explicit data reception callback.
        */
        @Override
        public void explicitDataReceived(ExplicitXBeeMessage xbeeMessage) {
                RemoteXBeeDevice remoteDevice = xbeeMessage.getDevice();
                int sourceEndpoint = xbeeMessage.getSourceEndpoint();
                int destEndpoint = xbeeMessage.getDestinationEndpoint();
                int clusterID = xbeeMessage.getClusterID();
                int profileID = xbeeMessage.getProfileID();
                String dataString = xbeeMessage.getDataString();
        }
}

[...]
```

To stop listening to new received explicit data, use the **removeExplicitDataListener (IExplicitDataReceiveListener)** method to unsubscribe the already-registered listener.

Data reception deregistration

```
[...]

ZigbeeDevice myZigbeeDevice = ...
MyExplicitDataReceiveListener myExplicitDataReceiveListener = ...
myZigbeeDevice.addExplicitDataListener(myExplicitDataReceiveListener);

[...]
```

```
// Remove the new explicit data reception listener.
myZigbeeDevice.removeExplicitDataListener(myExplicitDataReceiveListener);

[...]
```

Example: Receive explicit data via callback

The XBee Java Library includes a sample application that demonstrates how to subscribe to the explicit data reception service in order to receive explicit data. It can be located in the following path:

/examples/communication/explicit/ReceiveExplicitDataSample

Notes:

If your XBee module is configured to receive explicit data (**APIOutputMode.EXPLICIT** or **APIOutputMode.EXPLICIT_ZDO_PASSTHRU**) and another device sends non-explicit data, you receive an explicit message whose application layer field values are:

- Source endpoint: 0xE8
- Destination endpoint: 0xE8
- Cluster ID: 0x0011
- Profile ID: 0xC10

When an XBee module receives explicit data with these values, the message notifies both data reception callbacks (explicit and non-explicit) in case you have registered them. If you read the received data with the polling mechanism, you also receive the message through both methods.

Receive IP data

Some applications developed with the XBee Java Library may require modules to receive IP data.

Only Cellular, NB-IoT and Wi-Fi protocols support the transmission of IP data. This means you cannot receive IP data using a generic XBeeDevice object; you must use the protocol-specific XBee device objects CellularDevice, NBIoTDevice or WiFiDevice.

XBee Cellular and Wi-Fi modules operate the same way as other TCP/IP devices. They can initiate communications with other devices or listen for TCP or UDP transmissions at a specific port. In either case, you must apply any of the receive methods explained in this section in order to read IP data from other devices:

Listening for incoming transmissions

If the Cellular or Wi-Fi module operates as a server, listening for incoming TCP or UDP transmissions, you must start listening at a specific port, something similar to the bind operation of a socket. The XBee Java Library provides a method to listen for incoming transmissions:

Method	Description
startListening(int)	Starts listening for incoming IP transmissions in the provided port.

Listening for incoming transmissions

```
import com.digi.xbee.api.CellularDevice;
```

```
[...]

// Instantiate a Cellular device object.
CellularDevice myDevice = new CellularDevice("COM1", 9600);
myDevice.open();

// Listen for TCP or UDP transmissions at port 1234.
myDevice.startListening(1234);

[...]
```

The **startListening** method may fail for the following reasons:

- If the listening port provided is lesser than 0 or greater than 65535, the method throws an **IllegalArgumentException** exception.
- If there is a timeout setting the listening port, the method throws a **TimeoutException** exception .
- Errors that register as an **XBeeException**:
  - If the operating mode of the device is not **API** or **API_ESCAPE**, the method throws an **InvalidOperatingModeException**.
  - If the response of the listening port command is not valid, the method throws an **ATCommandException**.
  - If there is an error writing to the XBee interface, the method throws a generic **XBeeException**.

You can call the **stopListening** method to stop listening for incoming TCP or UDP transmissions:

Method	Description
stopListening()	Stops listening for incoming IP transmissions.

Stop listening for incoming transmissions

```
import com.digi.xbee.api.CellularDevice;

[...]

// Instantiate a Cellular device object.
CellularDevice myDevice = new CellularDevice("COM1", 9600);
myDevice.open();

// Stop listen for TCP or UDP transmissions.
myDevice.stopListening();

[...]
```

The **stopListening** method may fail for the following reasons:

- There is a timeout setting the listening port, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

Polling for data

The simplest way to read IP data is by executing the **readIPData** method of the local Cellular or Wi-Fi devices. This method blocks your application until IP data is received or the provided timeout has expired.

Method	Description
readIPData (int)	Specifies the time to wait in milliseconds for IP data reception (method blocks during that time or until IP data is received). If you don't specify a timeout, the method uses the default receive timeout configured in XBeeDevice.

Read network data (polling)

```
import com.digi.xbee.api.CellularDevice;
import com.digi.xbee.api.models.IPMessage;

[...]

// Instantiate a Cellular device object.
CellularDevice myDevice = new CellularDevice("COM1", 9600);
myDevice.open();

// Read IP data.
IPMessage ipMessage = myDevice.readIPData();

[...]
```

The method returns the read data inside a IPMessage object and contains the following information:

- IP address of the device that sent the data
- Transmission protocol
- Source and destination ports
- Byte array with the contents of the received data

You can retrieve the previous information using the corresponding getters of the **IPMessage** object:

Get the IPMessage information

```
import java.net.Inet4Address;
import com.digi.xbee.api.CellularDevice;
import com.digi.xbee.api.models.IPMessage;
import com.digi.xbee.api.models.IPProtocol;

[...]
```

```
// Instantiate a Cellular device object.
CellularDevice myDevice = [...]

// Read IP data.
IPMessage ipMessage = myDevice.readIPData();

Inet4Address destAddr = ipMessage .getIPAddress();
IPProtocol protocol = ipMessage .getProtocol();
int srcPort = ipMessage .getSourcePort();
int destPort = ipMessage .getDestPort();
byte[] data = ipMessage .getData();

[...]
```

You can also read IP data that comes from a specific IP address. For that purpose, the Cellular and Wi-Fi device objects provide the **readIPDataFrom** method:

Read network data from a specific remote XBee device (polling)

```
import java.net.Inet4Address;
import com.digi.xbee.api.CellularDevice;
import com.digi.xbee.api.models.IPMessage;

[...]

Inet4Address ipAddr = (Inet4Address) Inet4Address.getByName("52.36.102.96");

// Instantiate a Cellular device object.
CellularDevice myDevice = new CellularDevice("COM1", 9600);
myDevice.open();

// Read IP data.
IPMessage ipMessage = myDevice.readIPDataFrom(ipAddr);

[...]
```

This method also returns an **IPMessage** object containing the same information described before.

In either case, the default timeout to wait for data is two seconds. You can configure this timeout with the **getReceiveTimeout** and **setReceiveTimeout** methods of an XBee device class.

Get/set the timeout for synchronous operations

```
import com.digi.xbee.api.CellularDevice;

[...]

public static final int NEW_TIMEOUT_FOR_SYNC_OPERATIONS = 5 * 1000; // 5 seconds

CellularDevice myDevice = [...]

// Retrieving the configured timeout for synchronous operations.
System.out.println("Current timeout: " + myDevice.getReceiveTimeout() + "
milliseconds.");

[...]

// Configuring the new timeout (in milliseconds) for synchronous operations.
myDevice.setReceiveTimeout(NEW_TIMEOUT_FOR_SYNC_OPERATIONS);
```

```
[...]
```

Example: Receive IP data with polling

The XBee Java Library includes a sample application that demonstrates how to receive IP data using the polling mechanism. You can locate the example in the following path:

/examples/communication/ip/ConnectToEchoServerSample

IP data reception callback

This mechanism for reading IP data does not block your application. Instead, you can be notified when new IP data has been received if you have subscribed or registered with the IP data reception service by using the **addIPDataListener(IIPDataReceiveListener)** method.

Network data reception registration

```
import com.digi.xbee.api.CellularDevice;
import com.digi.xbee.api.listeners.IIPDataReceiveListener;

[...]

// Instantiate a Cellular device object.
CellularDevice myDevice = new CellularDevice("COM1", 9600);
myDevice.open();

// Subscribe to IP data reception.
myDevice.addIPDataListener(new MyIPDataReceiveListener());

[...]
```

The listener provided to the subscribed method, **MyIPDataReceiveListener**, must implement the **IIPDataReceiveListene**r interface. This interface includes the method that is executed when a new IP data is received by the XBee device.

When new IP data is received, the **ipDataReceived()** method of the **IIPDataReceiveListener** is executed providing as parameter an **IPMessage** object which contains the data and other useful information.

IPDataReceiveListener implementation example

```
import com.digi.xbee.api.listeners.IIPDataReceiveListener;
import com.digi.xbee.api.models.IPMessage;

public class MyIPDataReceiveListener implements IIPDataReceiveListener {
     /*
     * IP data reception callback.
     */
     @Override
     public void ipDataReceived(IPMessage ipMessage) {
          Inet4Address destAddr = ipMessage.getIPAddress();
          IPProtocol protocol = ipMessage.getProtocol();
          int srcPort = ipMessage.getSourcePort();
          int destPort = ipMessage.getDestPort();
          String dataString = ipMessage.getDataString();
          System.out.println("Received IP data from " + destAddr + ": " +
dataString);
```

```
    }
}
```

The IPMessage object provides the following information:

- IP address of the device that sent the data
- Transmission protocol
- Source and destination ports
- Byte array with the contents of the received data

You can retrieve the previous information using the corresponding getters of the **IPMessage** object.

To stop listening to new received IP data, use the **removeIPDataListener(IIPDataReceiveListener)** method to unsubscribe the already-registered listener.

Data reception deregistration

```
[...]

CellularDevice myDevice = ...
MyIPDataReceiveListener myipDataReceiveListener = ...

myDevice.addIPDataListener(myIPDataReceiveListener);

[...]

// Remove the IP data reception listener.
myDevice.removeIPDataListener(myIPDataReceiveListener);

[...]
```

Example: Receive IP data with listener

The XBee Java Library includes a sample application that demonstrates how to receive IP data using the listener. You can locate the example in the following path:

/examples/communication/ip/ReceiveIPDataSample

Receive IPv6 data

Some applications using IPv6 based devices require modules to receive IPv6 data

Only Thread protocol supports the reception of IPv6 data. This means you cannot receive IPv6 data using a generic XBeeDevice object; you must use the protocol-specific XBee device object ThreadDevice.

XBee Thread radio modules can initiate communications with other Thread devices in the network or listen for UDP transmissions at a specific port. You must apply any of the receive methods explained in this section to read IPv6 data from other Thread devices:

- Listening for incomming transmissions
- Polling for data
- IPv6 data reception callback

Listening for incomming transmissions

If the Thread module operates as a server listening for incoming UDP transmissions, you must start listening at a specific port, similarly to the bind operation of a socket. The XBee Java Library provides a method to listen for incoming transmissions:

Method	Description
startListening(int)	Starts listening for incoming UDP transmissions in the provided port.

Listening for incoming transmissions

```
import com.digi.xbee.api.ThreadDevice;

[...]

// Instantiate a Thread device object.
ThreadDevice myDevice = new ThreadDevice("COM1", 9600);
myDevice.open();

// Listen for UDP transmissions at port 1234.
myDevice.startListening(1234);

[...]
```

The **startListening** method may fail for the following reasons:

- If the listening port provided is lesser than 0 or greater than 65535, the method throws an **IllegalArgumentException**.
- If there is a timeout setting the listening port, the method throws a **TimeoutException**.
- Errors that register as an **XBeeException**:
  - If the operating mode of the device is not **API** or **API_ESCAPE**, the method throws an **InvalidOperatingModeException**.
  - The response of the listening port command is not valid, throwing an **ATCommandException**.
  - If there is an error writing to the XBee interface, the method throws a generic **XBeeException**.

You can call the **stopListening** method to stop listening for incoming UDP transmissions:

Method	Description
stopListening()	Stops listening for incoming UDP transmissions.

Stop listening for incoming transmissions

```
import com.digi.xbee.api.ThreadDevice;

[...]

// Instantiate a Thread device object.
ThreadDevice myDevice = new ThreadDevice("COM1", 9600);
```

```
myDevice.open();

// Stop listening for UDP.
myDevice.stopListening();

[...]
```

The **stopListening** method may fail for the following reasons:

- There is a timeout setting the listening port, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

Polling for data

The simplest way to read IPv6 data is by executing the **readIPData** method of the local Thread device. This method blocks your application until IPv6 data is received or the provided timeout has expired.

Method	Description
readIPData (int)	Specifies the time to wait in milliseconds for IPv6 data reception (method blocks during that time or until IPv6 data is received). If you don't specify a timeout, the method uses the default receive timeout configured in XBeeDevice.

Read IPv6 data (polling)

```
import com.digi.xbee.api.ThreadDevice;
import com.digi.xbee.api.models.IPMessage;

[...]

// Instantiate a Thread device object
ThreadDevice myDevice = new ThreadDevice("COM1", 9600);
myDevice.open();

// Read IPv6 data.
IPMessage ipMessage = myDevice.readIPData();

[...]
```

The method returns the read data inside a IPMessage object and contains the following information:

- IPv6 address of the device that sent the data
- Transmission protocol
- Source and destination ports
- Byte array with the contents of the received data
- You can retrieve the previous information using the corresponding getters of the **IPMessage** object:

Get the IPMessage information

```
import java.net.Inet6Address;
import com.digi.xbee.api.ThreadDevice;
import com.digi.xbee.api.models.IPMessage;
import com.digi.xbee.api.models.IPProtocol;

[...]

// Instantiate a Thread device object.
ThreadDevice myDevice = [...]

// Read IP data.
IPMessage ipMessage = myDevice.readIPData();

Inet6Address destAddr = ipMessage .getIPAddress();
IPProtocol protocol = ipMessage .getProtocol();
int srcPort = ipMessage .getSourcePort();
int destPort = ipMessage .getDestPort();
byte[] data = ipMessage .getData();

[...]
```

You can also read IPv6 data that comes from a specific IPv6 address. For that purpose, the Thread device objects provide the **readIPDataFrom** method:

Read network data from a specific remote XBee device (polling)

```
import java.net.Inet6Address;
import com.digi.xbee.api.ThreadDevice;
import com.digi.xbee.api.models.IPMessage;

[...]

Inet6Address ipAddr = (Inet6Address) Inet6Address.getByName
("FDB3:0001:0002:0000:0004:0005:0006:0007");

// Instantiate a Thread device object.
ThreadDevice myDevice = new ThreadDevice("COM1", 9600);
myDevice.open();

// Read IPv6 data.
IPMessage ipMessage = myDevice.readIPDataFrom(ipAddr);

[...]
```

This method also returns an **IPMessage** object containing the same information described before.

In either case, the default timeout to wait for data is two seconds. You can configure this timeout with the **getReceiveTimeout** and **setReceiveTimeout** methods of an XBee device class.

Get/set the timeout for synchronous operations

```
import com.digi.xbee.api.ThreadDevice;

[...]

public static final int NEW_TIMEOUT_FOR_SYNC_OPERATIONS = 5 * 1000; // 5 seconds
```

```
ThreadDevice myDevice = [...]

// Retrieving the configured timeout for synchronous operations.
System.out.println("Current timeout: " + myDevice.getReceiveTimeout() + "
milliseconds.");

[...]

// Configuring the new timeout (in milliseconds) for synchronous operations.
myDevice.setReceiveTimeout(NEW_TIMEOUT_FOR_SYNC_OPERATIONS);

[...]
```

IPv6 data reception callback

This mechanism for reading IPv6 data does not block your application. Instead, you can be notified when new IPv6 data has been received if you have subscribed or registered with the IPv6 data reception service by using the **addIPDataListener(IIPDataReceiveListener)** method.

Network data reception registration

```
import com.digi.xbee.api.ThreadDevice;
import com.digi.xbee.api.listeners.IIPDataReceiveListener;

[...]

// Instantiate a Thread device object.
ThreadDevice myDevice = new ThreadDevice("COM1", 9600);
myDevice.open();

// Subscribe to IPv6 data reception.
myDevice.addIPDataListener(new MyIPDataReceiveListener());

[...]
```

The listener provided to the subscribed method, **MyIPDataReceiveListener**, must implement the **IIPDataReceiveListener** interface. This interface includes the method that executes when a new IP data is received by the XBee device.

When new IP data is received, the **ipDataReceived()** method of the **IIPDataReceiveListener** is executed providing as parameter an **IPMessage** object which contains the data and other useful information.

IPDataReceiveListener implementation example

```
import java.net.Inet6Address;
import com.digi.xbee.api.listeners.IIPDataReceiveListener;
import com.digi.xbee.api.models.IPMessage;

public class MyIPDataReceiveListener implements IIPDataReceiveListener {
     /*
     * IP data reception callback.
     */
     @Override
     public void ipDataReceived(IPMessage ipMessage) {
          Inet6Address destAddr = ipMessage.getIPv6Address();
          IPProtocol protocol = ipMessage.getProtocol();
          int srcPort = ipMessage.getSourcePort();
          int destPort = ipMessage.getDestPort();
          String dataString = ipMessage.getDataString();
          System.out.println("Received IPv6 data from " + destAddr + ": " +
dataString);
     }
}
```

The IPMessage object provides the following information:

- IP address of the device that sent the data
- Transmission protocol
- Source and destination ports
- Byte array with the contents of the received data

You can retrieve the previous information using the corresponding getters of the **IPMessage** object.

To stop listening to new received IP data, use the **removeIPDataListener(IIPDataReceiveListener)** method to unsubscribe the already-registered listener.

Data reception deregistration

```
[...]

ThreadDevice myDevice = ...
MyIPDataReceiveListener myipDataReceiveListener = ...

myDevice.addIPDataListener(myIPDataReceiveListener);

[...]

// Remove the IP data reception listener.
myDevice.removeIPDataListener(myIPDataReceiveListener);

[...]
```

Example: Receive IPv6 data with listener

The XBee Java Library includes a sample application that demonstrates how to receive IPv6 data using the listener. You can locate the example in the following path:
/examples/communication/ip/ReceiveIPv6DataSample

Receive CoAP data

Received CoAP data is captured as IPv6 data, which means that you receive CoAP data the same as you receive IPv6 data. See Receive IPv6 data for more information.

Example: Receive CoAP data with listener

The XBee Java Library includes a sample application that demonstrates how to receive CoAP data using the listener. You can locate the example in the following path:

/examples/communication/coap/ReceiveCoAPDataSample

Receive SMS messages

Some applications developed with the XBee Java Library may require modules to receive SMS messages.

Only Cellular modules support the reception of SMS messages.

SMS reception callback

You can be notified when a new SMS has been received if you are subscribed or registered to the SMS reception service by using the **addSMSListenerListener(ISMSReceiveListener)** method.

SMS reception registration

```
import com.digi.xbee.api.CellularDevice;
import com.digi.xbee.api.listeners.ISMSReceiveListener;

[...]

// Instantiate a Cellular device object.
CellularDevice myDevice = new CellularDevice("COM1", 9600);
myDevice.open();

// Subscribe to SMS reception.
myDevice.addSMSListener(new MySMSReceiveListener());

[...]
```

The listener provided to the subscribed method, **MySMSReceiveListener**, must implement the **ISMSReceiveListener** interface. This interface includes the method that is executed when a new SMS is received by the XBee device.

When that occurs, the **smsReceived()** method of the **ISMSReceiveListener** is executed providing as parameter an **SMSMessage** object which contains the data and the phone number that sent the message. You can retrieve that information by using the corresponding getters.

SMSReceiveListener implementation example

```
import com.digi.xbee.api.listeners.ISMSReceiveListener;
import com.digi.xbee.api.models.SMSMessage;

public class MySMSReceiveListener implements ISMSReceiveListener {
     /*
     * SMS reception callback.
     */
     @Override
     public void smsReceived(SMSMessage smsMessage) {
         String phoneNumber = smsMessage.getPhoneNumber();
         String data = smsMessage.getData();
         System.out.println("Received SMS from " + phoneNumber + ": " + data);
     }
}
```

To stop listening to new SMS messages, use the **removeSMSListener(ISMSReceiveListener)** method to unsubscribe the already-registered listener.

SMS reception deregistration

```
[...]

CellularDevice myDevice = ...
MySMSReceiveListener mySMSReceiveListener = ...

myDevice.addSMSListener(mySMSReceiveListener);

[...]

// Remove the SMS reception listener.
myDevice.removeSMSListener(mySMSReceiveListener);

[...]
```

Example: Receive SMS messages

The XBee Java Library includes a sample application that demonstrates how to subscribe to the SMS reception service in order to receive text messages. You can locate the example in the following path:

/examples/communication/cellular/ReceiveSMSSample

Receive modem status events

A local XBee device is able to determine when it connects to a network, when it is disconnected, and when any kind of error or other events occur. The local device generates these events, and they can be handled using the XBee Java library through the modem status frames reception.

When a modem status frame is received, you are notified through the callback of a custom listener, so you can take the proper actions depending on the event received.

For that purpose, you must subscribe or register to the modem status reception service using a modem status listener as parameter with the method **addModemStatusListener (IModemStatusReceiveListener)**.

Modem status reception registration

```
import com.digi.xbee.api.XBeeDevice;

[...]

// Instantiate an XBee device object.
XBeeDevice myXBeeDevice = new XBeeDevice("COM1", 9600);
myXBeeDevice.open();

// Creation of Modem Status listener.
MyModemStatusListener myModemStatusListener = ...

// Subscribe to modem status events reception.
myXBeeDevice.addModemStatusListener(myModemStatusListener);

[...]
```

The listener to be subscribed, **MyModemStatusListener**, must implement the **IModemStatusReceiveListener** interface. This interface includes the method executed when a modem status event is received by the XBee device.

It does not matter the type of local XBee device you have instanced, as this data reception operation is implemented the same way for all the local XBee device protocols.

When a new modem status event is received, the **modemStatusEventReceived()** method of the **IModemStatusReceiveListener** is executed, providing a **ModemStatusEvent** enumeration entry object parameter, which contains the information about the event.

IModemStatusReceiveListener implementation example, MyModemStatusListener

```
import com.digi.xbee.api.models.ModemStatusEvent;
import com.digi.xbee.api.listeners.IModemStatusReceiveListener;

public class MyModemStatusListener implements IModemStatusReceiveListener {
       /*
       * Modem status event reception callback.
       */
       @Override
       public void modemStatusEventReceived(ModemStatusEvent modemStatusEvent) {
               System.out.println("Received modem status: " +
                               modemStatusEvent.toString());
       }
}
```

To stop listening to modem status events, use the **removeModemStatusListener (IModemStatusReceiveListener)** method.

Removing the modem status listener

```
[...]

XBeeDevice myXBeeDevice = ...
MyModemStatusListener myModemStatusListener = ...

myXBeeDevice.addModemStatusListener(myModemStatusListener);

[...]

// Remove the modem status listener.
myXBeeDevice.removeModemStatusListener(myModemStatusListener);

[...]
```

Modem status reception example

The XBee Java Library includes a sample application that shows you how to subscribe to the modem status reception service to receive modem status events. The example is located in the following path:

/examples/communication/ReceiveModemStatusSample

Handling analog and digital IO lines

All the XBee modules, regardless of the protocol they run, have a set of lines (pins). You can use these pins to connect sensors or actuators and configure them with specific behavior.

You can configure the IO lines of an XBee device to be digital input/output (DIO), analog to digital converter (ADC), or pulse-width modulation output (PWM). The configuration you provide to a line depends on the device where you want to connect.

Note All the IO management features displayed in this topic and sub-topics are applicable for both local and remote XBee devices.

The XBee Java Library exposes an easy way to configure, read, and write the IO lines of the local and remote XBee devices through the following corresponding classes:

- XBeeDevice for local devices.
- RemoteXBeeDevice for remotes.

This section provides information to show you how to complete the following tasks:

- Configure the IO lines
- Read IO samples

Configure the IO lines

All XBee device objects include a configuration method, **setIOConfiguration(IOLine, IOMode)**, where you can specify the IO line being configured and the desired function being set.

For the IO line parameter, the API provides an enumerator called **IOLine** that helps you specify the desired IO line easily by functional name. This enumerator is used along all the IO related methods in the API.

The supported functions are also contained in an enumerator called **IOMode**. You can choose between the following functions:

- DISABLED
- SPECIAL_FUNCTIONALITY (Shouldn't be used to configure IOs)
- PWM
- ADC
- DIGITAL_IN
- DIGITAL_OUT_LOW
- DIGITAL_OUT_HIGH

Configuring local or remote IO lines

```
import com.digi.xbee.api.RemoteXBeeDevice;
import com.digi.xbee.api.XBeeDevice;
import com.digi.xbee.api.io.IOLine;
import com.digi.xbee.api.io.IOMode;

[...]

// Instantiate a local XBee device object.
XBeeDevice myLocalXBeeDevice = new XBeeDevice("COM1", 9600);
myLocalXBeeDevice.open();

// Instantiate a remote XBee device object.
RemoteXBeeDevice myRemoteXBeeDevice = new RemoteXBeeDevice(myLocalXBeeDevice,
                                      new XBee64BitAddress("000000409D5EXXXX"));

// Configure the DIO1_AD1 line to be Digital output (set high by default).
myLocalXBeeDevice.setIOConfiguration(IOLine.DIO1_AD1, IOMode.DIGITAL_OUT_HIGH);

// Configure the DIO2_AD2 line to be Digital input.
myLocalXBeeDevice.setIOConfiguration(IOLine.DIO2_AD2, IOMode.DIGITAL_IN);

// Configure the DIO3_AD3 line to be Analog input (ADC).
myRemoteXBeeDevice.setIOConfiguration(IOLine.DIO3_AD3, IOMode.ADC);
```

```
// Configure the DIO10_PWM0 line to be PWM output (PWM).
myRemoteXBeeDevice.setIOConfiguration(IOLine.DIO10_PWM0, IOMode.PWM);

[...]
```

The **setIOConfiguration()** method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

You can read the current configuration of any IO line the same way an IO line can be configured with a desired function using the corresponding getter, **getIOConfiguration(IOLine)**.

Getting IOs configuration

```
[...]

// Get the configuration mode of the DIO1_AD1 line.
IOMode ioMode = myXBeeDevice.getIOConfiguration(IOLine.DIO1_AD1);

[...]
```

The **getIOConfiguration()** method may fail for the following reasons:

- ACK of the read command is not received in the configured timeout, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - If the operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - If the received response does not contain the value for the given IO line, throwing a **OperationNotSupportedException**.
  - If the response to the read command is not valid, throwing an **ATCommandException**.
  - If there is an error writing to the XBee interface, throwing a generic **XBeeException**.

Digital input/output

If your IO line is configured as digital output, you can set its state (high/low) easily. All the XBee device classes provide the method, **setDIOValue(IOLine, IOValue)**, with the desired IO line as the first parameter and an **IOValue** as the second. The IOValue enumerator includes HIGH and LOW as possible values.

Setting digital output values

```
[...]

// Set the DIO2_AD2 line low.
```

```
myXBeeDevice.setDIOValue(IOLine.DIO2_AD2, IOValue.LOW);

// Set the DIO2_AD2 line high.
myXBeeDevice.setDIOValue(IOLine.DIO2_AD2, IOValue.HIGH);

[...]
```

The **setDIOValue()** method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

You can also read the current status of the pin (high/low) by issuing the method **getDIOValue(IOLine)**. The parameter of the method must be the IO line to be read.

Reading digital input values

```
[...]

// Get the value of the DIO2_AD2.
IOValue value = myXBeeDevice.getDIOValue(IOLine.DIO2_AD2);

[...]
```

- ACK of the read command is not received in the configured timeout, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - If the operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - If the received response does not contain the value for the given IO line, throwing a **OperationNotSupportedException**. This can happen (for example) if you try to read the DIO value of an IO line that is not configured as Digital Input.
  - If the response to the read command is not valid, throwing an **ATCommandException**.
  - If there is an error writing to the XBee interface, throwing a generic **XBeeException**.

Handling DIO IO Lines example

The XBee Java Library includes two sample applications that demonstrate how to handle DIO lines in your local and remote XBee Devices. The examples are located in the following path:

/examples/io/LocalDIOSample

/examples/io/RemoteDIOSample

ADC

When you configure an IO line as analog to digital converter (ADC), you can only read its value (counts). In this case, the method used to read ADCs is different than the digital I/O method, but the parameter provided is the same. The IO line to read the value from **getADCValue(IOLine)**.

Reading ADC values

```
[...]

// Get the value of the DIO 3 (analog to digital converter).
int value = myXBeeDevice.getADCValue(IOLine.DIO3_AD3);

[...]
```

The **getADCValue()** method may fail for the following reasons:

- ACK of the read command is not received in the configured timeout, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
    - If the operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
    - If the received response does not contain the value for the given IO line, throwing a **OperationNotSupportedException**. This can happen (for example) if you try to read the ADC value of an IO line that is not configured as ADC.
    - If the response to the read command is not valid, throwing an **ATCommandException**.
    - If there is an error writing to the XBee interface, throwing a generic **XBeeException**.

Handling ADC IO Lines example

The XBee Java Library includes two sample applications that demonstrate how to handle ADC lines in your local and remote XBee Devices. The examples are located in the following path:
/examples/io/LocalADCSample
/examples/io/RemoteADCSample

PWM

Not all the XBee protocols support pulse-width modulation (PWM) output handling, but the XBee Java Library provides functionality to manage them. When you configure an IO line as PWM output, you must use specific methods to set and read the duty cycle of the PWM.

For the set case, use the method **setPWMDutyCycle(IOLine, double)** and provide the IO line configured as PWM and the value of the duty cycle in % of the PWM. The duty cycle is the proportion of 'ON' time to the regular interval or 'period' of time. A high duty cycle corresponds to high power, because the power is ON for most of the time. The percentage parameter of the set duty cycle method is a double, which allows you to be more precise in the configuration.

Setting the duty cycle of an IO line configure as PWM

```
[...]

// Set a duty cycle of 75% to the DIO10_PWM0 line (PWM output).
myXBeeDevice.setPWMDutyCycle(IOLine.DIO10_PWM0, 75);

[...]
```

The **setPWMDutyCycle()** method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

The **getPWMDutyCycle(IOLine)** method of a PWM line returns a double value with the current duty cycle percentage of the PWM.

Getting the duty cycle of an IO line configured as PWM

```
[...]

// Get the duty cycle of the DIO10_PWM0 line (PWM output).
double dutyCycle = myXBeeDevice.getPWMDutyCycle(IOLine.DIO10_PWM0);

[...]
```

The **getPWMDutyCycle()** method may fail for the following reasons:

- ACK of the read command is not received in the configured timeout, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - If the operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - If the received response does not contain the value for the given IO line, throwing a **OperationNotSupportedException**.
  - If the response to the read command is not valid, throwing an **ATCommandException**.
  - If there is an error writing to the XBee interface, throwing a generic **XBeeException**.

Note In both cases (get and set), the IO line provided must be PWM capable and must be configured as PWM output.

Read IO samples

XBee modules have the ability to monitor and sample the analog and digital IO lines. You can read IO samples locally or transmitted to a remote device to provide an indication of the current IO line states.

There are three ways to obtain IO samples on a local or remote device:

- Queried sampling
- Periodic sampling
- Change detection sampling

The XBee Java Library represents an IO sample by the **IOSample class**, which contains:

- Digital and analog channel masks that indicate which lines have sampling enabled.
- Values of those enabled lines.

You must configure the IO lines you want to receive in the IO samples before enabling sampling.

Queried sampling

The XBee Java Library provides a method to read an IO sample that contains all enabled digital IO and analog input channels, **readIOSample()**. The method returns an **IOSample** object.

Reading an IO sample and getting the DIO value

```
[...]

// Read an IO sample from the device.
IOSample ioSample = myXBeeDevice.readIOSample();

// Select the desired IO line.
IOLine ioLine = IOLine.DIO3_AD3;

// Check if the IO sample contains the expected IO line and value.
if (ioSample.hasDigitalValue(ioLine)) {
       System.out.println(ioSample.getDigitalValue(ioLine));
}

[...]
```

The **readIOSample()** method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

Periodic sampling

Periodic sampling allows an XBee module to take an IO sample and transmit it to a remote device at a periodic rate. That remote device is defined in the destination address through the **setDestinationAddress(XBee64BitAddress)** method. The XBee Java Library provides the **setIOSamplingRate(int)** method to configure the periodic sampling.

The XBee module samples and transmits all enabled digital IO and analog inputs to the remote device every X milliseconds. A sample rate of 0 ms disables this feature.

Setting the IO sampling rate

```
[...]

// Set the destination address.
myXBeeDevice.setDestinationAddress(new XBee64BitAddress("0013A20040XXXXXX"));

// Set the IO sampling rate.
myXBeeDevice.setIOSamplingRate(5000); // 5 seconds.

[...]
```

The **setIOSamplingRate()** method may fail for the following reasons:

- The sampling rate is lower than 0, throwing an **IllegalArgumentException**.
- ACK of the command sent is not received in the configured timeout, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

You can also read this value using the **getIOSamplingRate()** method. This method returns the IO sampling rate in milliseconds and '0' when the feature is disabled.

Getting the IO sampling rate

```
[...]

// Get the IO sampling rate.
int value = myXBeeDevice.getIOSamplingRate();

[...]
```

The **getIOSamplingRate()** method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

Change detection sampling

You can configure modules to transmit a data sample immediately whenever a monitored digital IO pin changes state. The **setDIOChangeDetection(Set<IOLine>)** method establishes the set of digital IO lines that are monitored for change detection. A null set disables the change detection sampling.

As in the periodic sampling, change detection samples are transmitted to the configured destination address.

Note This feature only monitors and samples digital IOs, so it is not valid for analog lines.

Setting the DIO change detection

```
[...]

// Set the destination address.
myXBeeDevice.setDestinationAddress(new XBee64BitAddress("0013A20040XXXXXX"));

// Create a set of IO lines to be monitored.
Set<IOLine> lines = EnumSet.of(IOLine.DIO3_AD3, IOLine.DIO4_AD4);
```

```
// Enable the DIO change detection sampling.
myXBeeDevice.setDIOChangeDetection(lines);

[...]
```

The **setIOSamplingRate()** method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

You can also get the lines that are monitored using the **getDIOChangeDetection()** method. A **null** indicates that this feature is disabled.

Getting the DIO change detection

```
[...]

// Get the set of lines that are monitored.
Set<IOLine> lines = myXBeeDevice.getDIOChangeDetection();

[...]
```

The **getDIOChangeDetection()** method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a **TimeoutException**.
- Other errors caught as **XBeeException**:
  - The operating mode of the device is not **API** or **API_ESCAPE**, throwing an **InvalidOperatingModeException**.
  - The response of the command is not valid, throwing an **ATCommandException**.
  - There is an error writing to the XBee interface, throwing a generic **XBeeException**.

Register an IO sample listener

In addition to configuring an XBee device to monitor and sample the analog and digital IO lines, you must register a listener in the local device where you want to receive the IO samples. You are then notified when the device receives a new IO sample.

You must subscribe to the IO samples reception service by using the method **addIOSampleListener (IIOSampleReceiveListener)** with an IO sample reception listener as parameter.

Registering an IO sample receive listener

```
[...]

// Create the IO sample listener.
MyIOSampleReceiveListener myIOSampleReceiveListener = ...
```

```
// Subscribe to IO samples reception.
myXBeeDevice.addIOSampleListener(myIOSampleReceiveListener);

[...]
```

The listener provided to the subscribe method, **MyIOSampleReceiveListener**, must implement the **IIOSampleReceiveListener** interface. This interface includes the method that executes when the XBee device receives a new IO sample.

Note This listener can only be registered on local devices, but is implemented the same way for all the XBeeDevice subclasses.

When the XBee device receives a new IO sample, the **ioSampleReceived()** method of the **IIOSampleReceiveListener** executes, providing as parameters a **RemoteXBeeDevice** object, which indicates the device that sent the sample, and an IOSample object with the IO data sample.

IIOSampleReceiveListener implementation example, MyIOSampleReceiveListener

```
import com.digi.xbee.api.listeners.IIOSampleReceiveListener;

public class MyIOSampleReceiveListener implements IIOSampleReceiveListener {
       @Override
       public void ioSampleReceived(RemoteXBeeDevice remoteDevice, IOSample ioSample) {
               System.out.println("IO sample from " + remoteDevice.get64BitAddress() +
                               " - " + ioSample.toString());
       }
}
```

To stop receiving notifications of new IO samples, use the **removeIOSampleListener (IIOSampleReceiveListener)** method.

Removing the modem status listener.

```
[...]

XBeeDevice myXBeeDevice= ...
MyIOSampleReceiveListener myIOSampleReceiveListener = ...

myXBeeDevice.addIOSampleListener(myIOSampleReceiveListener );

[...]

// Remove the IO sample listener.
myXBeeDevice.removeIOSampleListener(myIOSampleReceiveListener);

[...]
```

IO Sampling example

The XBee Java Library includes a sample application that demonstrates how to configure a remote device to monitor IO lines and receive the IO samples in the local device. The example is located in the following path:

/examples/io/IOSamplingSample

Logging events

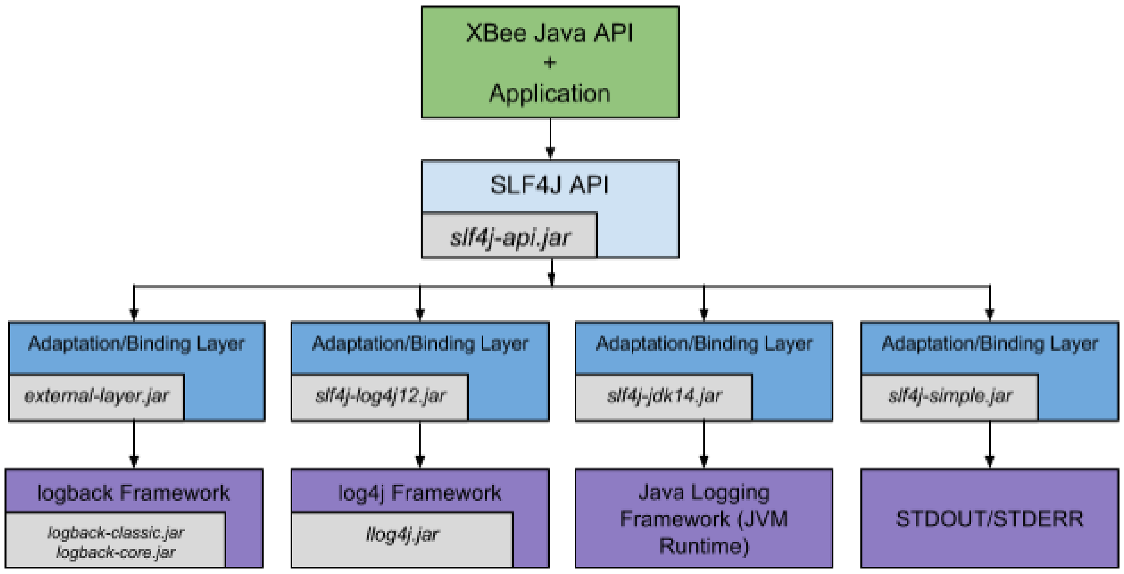
Logging is a fundamental part of applications, and every application includes this feature. A well-designed logging system is a useful utility for system administrators, developers and the support team and can save valuable time in sorting through the cause of issues. As users execute programs at the front end, the system invisibly builds a vault of event information (log entries) for system administrators and the support team.

There are many available logging libraries and logging frameworks for Java, but this API does not force users to use any specific one. The XBee Java Library is built on top of the **Simple Logging Facade for Java (SLF4J)**. For more information about SLF4J, see http://www.slf4j.org/.

SLF4J serves as a simple facade or abstraction for various logging frameworks (for example, **java.util.logging, logback, log4j**) allowing the end user to plug in the desired logging framework at deployment time in the final application.

Most of the important open source projects at the moment use this abstraction layer to let users decide on their final logging implementation strategy. It is also very common to use it in APIs and libraries that are later integrated in a user application.

SLF4J does not use a specific logging framework. To use a logging framework you must include a binding library in your application classpath as well as the final logger library (if applicable).



The only library that is required for use of the logging features in the XBee Java Library is the slf4j-api.jar, which has a simple syntax for logging messages. The SLF4J API exposes all the required methods and functions to log messages from the XBee Java Library and from the final user application. It then relies on a binding library that is specific for each underlying logging framework to be included by users in the final application.

This section provides information to show you how to complete the following tasks:

- Download the SLF4J bindings
- Bind the library with SLF4J

Download the SLF4J bindings

You can download the latest SLF4J version 1.7.12 including full source code, binding libraries, class files and documentation in ZIP or TAR.GZ format from this location:

http://www.slf4j.org/download.html

Bind the library with SLF4J

As mentioned previously, SLF4J supports various logging frameworks. The SLF4J distribution ships with several jar files referred to as SLF4J bindings, with each binding corresponding to a supported logging framework:

Binding Jar	Logging framework
slf4j-log4j12-1.7.12.jar	log4j version 1.2
slf4j-jdk14-1.7.12.jar	**java.util.logging** (built in Java logging framework)
slf4j-nop-1.7.12.jar	NOP, silently discards all logging
slf4j-simple-1.7.12.jar	Simple implementation, which outputs all events to **System.err**
slf4j-jcl-1.7.12.jar	Jakarta Commons Logging
slf4j-android-1.7.12.jar	Android logging
external implementations	Support for other logging frameworks, for example logback

If no SLF4J binding is found in your application classpath, SLF4J defaults to a no-operation implementation displaying the following output in the console:

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further
details.
```

If you want to use a specific logging framework in your application, you must include the specific logging library as well as the corresponding binding library in your classpath. SLF4J automatically detects the binding library and calls the corresponding underlying logging framework, so you only have to care for the upper layer.

Building the library

To build the XBee Java Library and execute the unit tests included, you can use Ant scripts, an IDE, or any other tool you prefer. The following software components are required for that purpose:

- RxTx 2.2 serial communication library (download link)
- Simple Logging Facade for Java (SLF4J) 1.7.12 library and your preferred logging library
- JUnit 4.x
- Mockito 1.10.19
- PowerMock 1.6.2

Note The API already includes support to use Apache Maven 3.x, allowing you to easily create the JAR file, execute the unit tests, build the samples, and launch them without the necessity of downloading all the requirements.

This section provides information to show you how to complete the following tasks:

- Install Apache Maven
- Install the library in Maven local repository

Install Apache Maven

Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.

To build the XBee API using Maven, complete the following steps:

1. Download the XBee Java Library sources from GitHub.
    a. Check out the project from GitHub or download a zip file by clicking the **Download ZIP** button available in the GitHub repository main page: https://github.com/digidotcom/XBeeJavaLibrary.
    b. If you download the zip file, you must uncompress the file. The resulting directory is **XBeeJavaLibrary-master** and contains all the project files and directories.
2. Install Apache Maven.
    a. Download Apache Maven from http://maven.apache.org/download.cgi.

    Note Maven 3.2 requires JDK 1.6 or above.

    b. Follow the instructions at http://maven.apache.org/install.html to install Apache Maven.

Install the library in Maven local repository

Maven allows you to complete the following tasks:

- Build the library and samples
- Execute the unit tests
- Create a JAR package
- Launch a sample

You can complete the tasks one by one or all at once by executing one command inside the root directory of the repository (where the main pom.xml (Project Object Model) is located):

1. Open a console session and change into the XBee Java Library directory, where the pom.xml is located.

```
#> cd XBeeJavaLibrary-master
```

2. Execute the following command:

```
#> mvn clean install
```

Note The “clean” portion is optional. It cleans up artifacts created by prior builds.

Build the library and samples

The main Project Object Model (POM) file (pom.xml) is located in the root directory of the repository. To build the library and its samples you only need to:

1. Open a console session and change into the XBee Java Library directory, where the pom.xml is located.

```
#> cd XBeeJavaLibrary-master
```

2. Execute the following command to build the sources and tests of the project.

```
#> mvn clean compile
```

Note The first time you execute this (or any other) command, Maven downloads all the plugins and related dependencies required to fulfill the command. From a clean installation of Maven, this can take while. If you execute the command again, Maven now has all the required downloads and can execute the command more quickly.

Maven cleans any previous build results to start a fresh new build of the API sources. After you execute this command, the following output appears:

```
#> mvn clean compile
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Build Order:
[INFO]
[INFO] XBee Java Library Project
[INFO] XBee Java Library
```

```
[INFO] XBee Java Library Distribution

[...]

[INFO]
[INFO] Using the builder
org.apache.maven.lifecycle.internal.builder.singlethreaded.SingleThreadedBuilder
with a thread count of 1
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building XBee Java Library Project 1.0.1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ xbeeapi-parent ---
[INFO] Deleting C:\Store\GIT\xbee\XBeeJavaLibrary\target

[...]

[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] XBee Java Library Project ......................... SUCCESS [  0.284 s]
[INFO] XBee Java Library ................................. SUCCESS [  1.995 s]
[INFO] XBee Java Library Distribution .................... SUCCESS [  0.002 s]

[...]
```

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3.500 s
[INFO] Finished at: 2014-12-10T12:51:54+01:00
[INFO] Final Memory: 22M/53M
[INFO] ------------------------------------------------------------------------
#>
```

The resulting class files are located inside the directory called target in the root of the repository.

Execute the unit tests

After you have successfully built the library sources, there are some unit tests to compile and execute.

In a console session and in the XBee Java API directory execute the following command:

```
#> mvn clean test
```

Maven downloads more dependencies this time. These are the dependencies and plugins necessary for executing the tests. Before compiling and executing the tests, Maven compiles the main code. That is, you only need to execute this command to compile the sources, compile the unit tests, and execute the tests.

The resulting class files are located inside the directory called **targe**t in the root of the repository.

```
#> mvn clean test
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------

[...]

[INFO] ------------------------------------------------------------------------
[INFO] Building XBee Java Library Project 1.0
[INFO] ------------------------------------------------------------------------
[INFO]

[...]

-------------------------------------------------------
T E S T S
-------------------------------------------------------
Running com.digi.xbee.api.ApplyChangesTest
```

```
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.248 sec
```

```
Running com.digi.xbee.api.ExecuteParameterTest
```

```
Tests run: 11, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec
```

```
Running com.digi.xbee.api.ForceDisassociateTest
```

```
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec

[...]

Results :

Tests run: 946, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] ------------------------------------------------------------------------
```

```
[INFO] Building Receive Modem Status Sample 1.0
[INFO] ------------------------------------------------------------------------
[INFO]

[...]

[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] XBee Java Library Project ......................... SUCCESS [  0.296 s]
[INFO] XBee Java Library ................................. SUCCESS [ 29.995 s]
[INFO] Receive Modem Status Sample ....................... SUCCESS [  0.078 s]

[...]

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 31.353 s
[INFO] Finished at: 2014-11-03T11:27:25+01:00
[INFO] Final Memory: 22M/60M
[INFO] ------------------------------------------------------------------------
#>
```

Create a JAR package

Making a JAR file is fairly simple and can be accomplished by executing the following command inside the XBee Java Library directory:

```
#> mvn clean package
```

The command compiles all the sources, executes the unit tests, and creates a JAR file for the library and every sample inside the target directory in the root of the the XBee Java Library folder.

Launch a sample

You can test the compiled examples. The classpath must include all the libraries needed by any of the samples, that are all located in your local Maven repository:

```
#> java -Djava.library.path=target\rxtx-native-libs -cp
target\examples\communication\
SendBroadcastDataSample\send-broadcast-data-sample-0.1-SNAPSHOT.jar;target\
library\xbeejapi-0.1-SNAPSHOT.jar;<local_maven_repo_path>\org\rxtx\rxtx\2.2\
rxtx-2.2.jar;<local_maven_repo_path>\org\slf4j\slf4j-api\1.7.7\slf4j-api-
1.7.7.jar;
<local_maven_repo_path>\org\slf4j\slf4j-jdk14\1.7.7\
slf4j-jdk14-1.7.7.jar com.digi.xbee.api.sendbroadcastdata.MainApp
```

Where **<local_maven_repo_path>** is the path to your local repository, by default **~/.m2/repository**.

Or use Maven to test the compiled examples:

1. In a console session and in the XBee Java Library directory, execute the following command to install the artifact you've generated (the JAR file) in your local repository:

```
#> mvn install
```

   Maven compiles all the sources, executes the unit tests, and creates a JAR file for the library and every sample inside the target directory in the root of the the XBee Java Library folder.

2. Go to the desired example directory, where the **pom.xml** of the sample is located, and locate the **Send Broadcast Data** sample.

```
#> cd examples/communication/SendBroadcastDataSample
```

3. Execute the following command to execute the application.

```
#> mvn exec:exec
[INFO] Scanning for projects...
[INFO]
[INFO] Using the builder
org.apache.maven.lifecycle.internal.builder.singlethreaded.SingleThreadedBuilder
with a thread
count of 1
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Send Broadcast Data Sample 1.0.1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- exec-maven-plugin:1.3.2:exec (default-cli) @ receive-broadcast-data-
sample ---
  +-----------------------------------------------+
  |  XBee Java Library Send Broadcast Data Sample  |
  +-----------------------------------------------+

nov 03, 2014 11:34:46 AM com.digi.xbee.api.XBeeDevice open
INFO: [COM1 - 9600/8/N/1/N] Opening the connection interface...
WARNING:  RXTX Version mismatch
        Jar version = RXTX-2.2pre1
        native lib Version = RXTX-2.2pre2
nov 03, 2014 7:34:47 AM com.digi.xbee.api.XBeeDevice open
INFO: [COM1 - 9600/8/N/1/N] Connection interface open.
Sending broadcast data: 'Hello XBee World!'...
Success
nov 03, 2014 11:34:47 AM com.digi.xbee.api.XBeeDevice close
INFO: [COM1 - 9600/8/N/1/N] 0013A2004055BB5E (XBPRO900 232 Adapter) - Connection
interface closed.
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.857 s
[INFO] Finished at: 2014-11-03T11:34:47+01:00
[INFO] Final Memory: 6M/15M
[INFO] ------------------------------------------------------------------------
```

Android integration

The XBee Java Library has full support for the Android OS. You can import the library when developing an Android application and use it to communicate with XBee radio modules connected to your Android device, as you would do with a PC. The main difference is that you must use a different serial port interface when instantiating an **XBeeDevice** object.

This documentation explains the Android serial port interfaces supported by the XBee Java Library used to instantiate an **XBeeDevice** object and how to create XBee Android applications.

Instantiate an XBee device object in Android

To work with the XBee Java Library in Android, you must instantiate the XBeeDevice objects differently than you typically would with a PC. The serial port interfaces used in Android need the context of the Android application. The XBee Java Library is compatible with the following Android serial port interfaces:

USB host serial port

One serial interface that is common for all the Android devices is the USB host serial port. This interface is usually found in the Android devices as a micro USB connector. To communicate with XBee radio modules connected through this interface you need to instantiate XBeeDevice objects providing the following parameters:

- Android context
- Serial port baud rate

Instantiating an XBeeDevice in Android - USB host

```
import com.digi.xbee.api.XBeeDevice;

[...]

// Instantiate an XBee device object connected to the USB host interface of
Android.
XBeeDevice myXBeeDevice = new XBeeDevice(context, 9600);

[...]
```

There is another constructor that allows you to specify an Android USB permission listener. This listener is notified when the user grants USB permissions to the application where XBee Java Library is included.

Instantiating an XBeeDevice in Android - USB host with permission listener

```
import com.digi.xbee.api.XBeeDevice;
import com.digi.xbee.api.connection.android.AndroidUSBPermissionListener;

[...]

// Instantiate an Android USB permissions listener.
AndroidUSBPermissionListener permissionListener = new
AndroidUSBPermissionListener() {
     @Override
     public void permissionReceived(boolean permissionGranted) {
          if (permissionGranted)
               System.out.println("User granted USB permission.");
          else
               System.out.println("User rejected USB permission.");
     }
};

[...]

// Instantiate an XBee device object connected to the USB host interface of
Android with permission listener.
XBeeDevice myXBeeDevice = new XBeeDevice(context, 9600, permissionListener);

[...]
```

Note All the protocol specific classes derived from XBeeDevice, such as ZigbeeDevice, DigiMeshDevice, have these constructors for the USB host serial port for Android as well.

Digi serial port

The Digi Embedded for Android devices, such as the ConnectCore 6 SBC, have a serial port interface that you can use to communicate with XBee radio modules connected to the XBee socket. For this interface you must instantiate the XBeeDevice objects providing the following parameters:

- Android context
- Serial port name (usually prepended by "/dev/tty*")
- Serial port baud rate

Instantiating an XBeeDevice in Android - Digi serial port

```
import com.digi.xbee.api.XBeeDevice;

[...]

// Instantiate an XBee device object connected to the XBee socket of a Digi
Embedded for Android device.
XBeeDevice myXBeeDevice = new XBeeDevice(context, "/dev/ttymxc4", 9600);

[...]
```

Note All the protocol specific classes derived from XBeeDevice such as ZigbeeDevice or DigiMeshDevice, also have this constructor for the Digi serial port.

Create an XBee Android application

The process to develop an XBee application for Android is similar to creating an application for Android devices. An additional step is referencing the XBee Java Library in your Android project in order to use the classes and methods that it provides.

For this tutorial you will use Android Studio, which is the official IDE to create, build, and debug applications for Android devices. This guide shows you how to create an empty application, link the XBee Java Library and instantiate an XBeeDevice object, as well as import an already developed example that uses the XBee Java Library to communicate with XBee devices connected to the Android device.

Create an XBee Android application from scratch

Before creating the Android project, download and unzip the latest version of the XBee Java Library (see XBee Java Library software). It contains the library JAR file, **xbee-java-library-X.Y.Z.jar**, and other necessary resources in the directory called extra-libs. The XBee Java Library depends on the following JAR files to work properly in Android:

- *android-sdk-addon-3.jar: Digi SDK Add-on for Android, which allows you to create apps for Digi Embedded devices.*
- *slf4j-api-1.7.12.jar: The Simple Logging Facade for Java (SLF4J) for logging.*

Open Android Studio and follow these steps to create an XBee Android application:

1. Create a new application with Android Studio using the **Empty Activity** template. For instructions, see the Android Developers Guide.
2. Copy the **xbee-java-library-X.Y.Z.jar** and **slf4j-api-1.7.12.jar** files to the **app/libs** directory of the project.

3. For each jar file copied in the app/libs directory:
   - Right click the jar file and select the **Add As Library...** option.
   - Press **OK** to link the library to the module.
4. Replace the application's main Activity code with the following code:

XBee Android application code

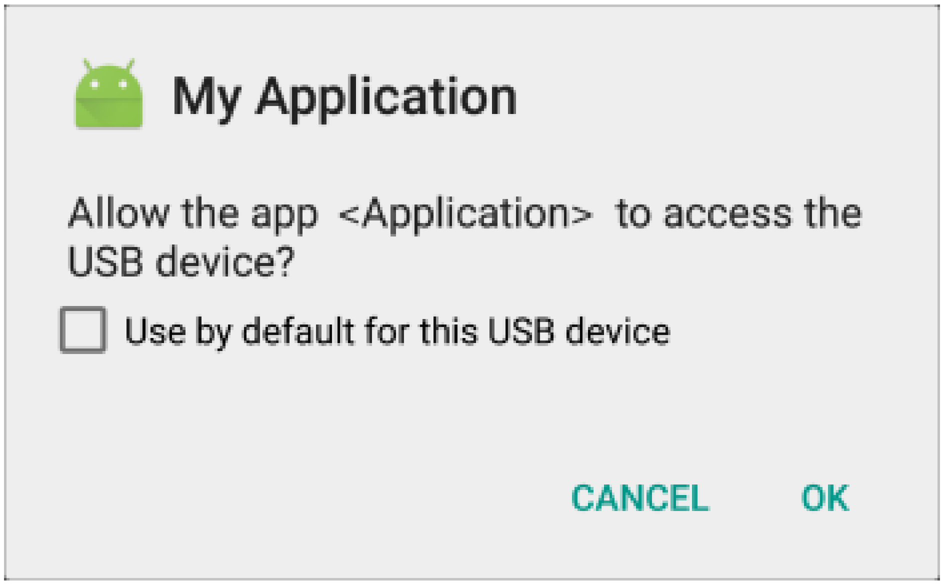
```
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.Toast;

import com.digi.xbee.api.XBeeDevice;
import com.digi.xbee.api.exceptions.XBeeException;

public class MainActivity extends AppCompatActivity {
     // Variables.
     private boolean connecting = false;
     private XBeeDevice myXBeeDevice = null;

     @Override
     protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
     }

     @Override
     protected void onResume() {
        super.onResume();

        // Avoid accesses while connecting.
        if (connecting)
           return;

        // Instantiate the XBeeDevice object.
        if (myXBeeDevice == null)
           myXBeeDevice = new XBeeDevice(this, 9600);

        // Create the connection thread.
        Thread connectThread = new Thread(new Runnable() {
           @Override
           public void run() {
              connecting = true;
              try {
                 // Check connection status.
                 if (myXBeeDevice.isOpen())
                    myXBeeDevice.close();
                 // Open device connection
                 myXBeeDevice.open();
                 showToastMessage("Device open: " + myXBeeDevice.toString());
              } catch (XBeeException e) {
                 showToastMessage("Could not open device: " + e.getMessage());
              }
              connecting = false;
           }
        });
        connectThread.start();
     }
     /**
     * Displays the given message.
     *
     * @param message The message to show.
     */
    private void showToastMessage(final String message) {
       runOnUiThread(new Runnable() {
          @Override
          public void run() {
             Toast.makeText(MainActivity.this, message, Toast.LENGTH_LONG).show
();
          }
       });
    }
}
```

This code tries to open the communication with the XBee radio module connected to the USB host port of the Android device every time the application resumes. If it succeeds, the application displays a toast message with the information of the XBee module. Otherwise, it displays a "Could not open device" message followed by the error that occurred.

5. To build the Android application, select **Build** > **Make Project** from the main menu bar.
6. To launch the application, select **Run** > **Run 'app'**.

Note This first time the application will fail to open the communication with the XBee radio module because the USB host interface is being used to communicate with the PC.

7. Remove the micro USB cable that connects the Android device to your PC, and connect the XBee module to the USB host interface of the Android device.
8. Resume the application from the Android UI, and grant permissions to access the USB device when prompted.


My Application

Allow the app <Application> to access the USB device?

☐ Use by default for this USB device

9. Verify the device can be found and the application displays its information in a toast message.

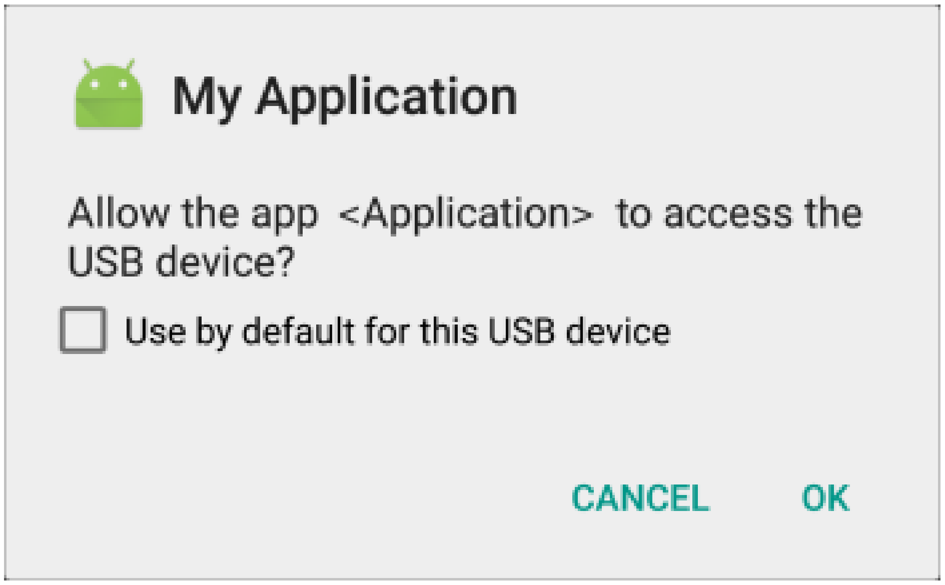
Import the XBee manager sample application

The XBee manager application demonstrates the usage of the XBee Java API in Android by providing an example of all the available options using a local XBee module attached to an Android device.

Before importing the sample in Android Studio, configure the **ANDROID_HOME** environment variable based on the location of the Android SDK and make sure you have Git installed.

Open Android Studio and complete the following steps to import the XBee manager sample application:

1. Version **Control** > **Git**.
2. Select **File** > **New** > **Project from Version Control** > **Git**.
3. On the **Clone Repository** dialog complete these fields with the following values:
   - **Git Repository URL**: https://github.com/digidotcom/XBeeManagerSample.
   - **Parent Directory**: Path to the folder that will host the repository directory.
   - **Directory Name**: Name of the repository directory.
4. Click **Clone** to import the example, and choose the **New Window** option if prompted about how to open the project.
5. Wait a few seconds while the compiler is working, and select the **Run** > **Run 'app'** option from the main menu bar to launch the application on the Android device.
6. When the application has launched, remove the micro USB cable that connects the Android device to your PC, and connect the XBee module to the USB host interface of the Android device.
7. Press the **Connect** button of the application and grant permissions to access the USB device when prompted.



My Application

Allow the app <Application> to access the USB device?

☐ Use by default for this USB device

The application allows you to communicate and configure the XBee module connected to your Android device and other remote XBee modules operating in the same network as the local module. When you open the connection with the local XBee module, the application layout changes and three new tabs are displayed:

- **XBee Device Info**: Displays information about the attached XBee module. You can configure some parameters from this tab.
- **Remote XBee Device**s: Discovers XBee modules in the same network as your local XBee module. You can select a remote device, change some parameters, and send data to it.
- **Received XBee Data**: Displays a table with received data from other XBee modules in the network.

XBee Java samples

The XBee Java Library includes several samples to demonstrate how to do the following:

- Communicate with your modules
- Configure your modules
- Read the IO lines
- Perform other common operations

All of the sample applications are contained in the examples folder of the **XBJL-X.Y.Z**, organized in categories. Every sample includes the source code and a **ReadMe** file to clarify the purpose and the required setup to launch the application.

Configuration samples

Manage common parameters

This sample Java application shows how to get and set common parameters of the XBee device. Common parameters are split in cached and non-cached parameters. For that reason, the application refreshes the cached parameters before reading and displaying them. The application then configures, reads, and displays the value of non-cached parameters.

The application uses the specific setters and getters provided by the XBee device object to configure and read the different parameters.

You can locate the example in the following path:

examples/configuration/ManageCommonParametersSample

Note For more information about how to manage common parameters, see Read and set common parameters.

Set and get parameters

This sample Java application shows how to set and get parameters of a local or remote XBee device. Use this method when you need to set or get the value of a parameter that does not have its own getter and setter within the XBee device object.

The application sets the value of four parameters with different value types:

- String
- Byte
- Array
- Integer

The application then reads the parameters from the device to verify that the read values are the same as the values that were set.

You can locate the example in the following path:
examples/configuration/SetAndGetParametersSample

Note For more information about how to get and set other parameters, see Read, set and execute other parameters.

Reset

This sample Java application shows how to perform a software reset on the local XBee module.

You can locate the example in the following path:

examples/configuration/ResetModuleSample

Note For more information about how to reset a module, see Reset the device.

Connect to access point (Wi-Fi devices)

This sample Java application shows how to configure a Wi-Fi module to connect to a specific access point and read its addressing settings.

You can locate the example at the following path:

examples/configuration/ConnectToAccessPoint

For more information about connecting to an access point, see Configure Wi-Fi settings.

Network samples - discover devices

This sample Java application demonstrates how to obtain the XBee network object from a local XBee device and discover the remote XBee devices that compose the network. The example adds a discovery listener, so the callbacks provided by the listener object receive the events.

The remote XBee devices are printed out as soon as they are found during the discovery.

You can locate the example in the following path:

examples/network/DiscoverDevicesSample

Note For more information about how to perform a network discovery, see Discover the network.

Communication samples

Send data

This sample Java application shows how to send data from the XBee device to another remote device on the same network using the XBee Java Library. In this example, the application sends data using a reliable transmission method. The application blocks during the transmission request, but you are notified if there is any error during the process.

The application sends data to a remote XBee device on the network with a specific node identifier (name).

You can locate the example in the following path:

examples/communication/SendDataSample

Note For more information about how to send data, see Send data.

Send data asynchronously

This sample Java application shows how to send data asynchronously from the XBee device to another remote device on the same network using the XBee Java Library. Transmitting data asynchronously means the execution is not blocked during the transmit request, but you cannot determine if the data was sent successfully.

The application sends data asynchronously to a remote XBee device on the network with a specific node identifier (name).

You can locate the example in the following path:

examples/communication/SendDataAsyncSample

Note For more information about how to get and set other parameters, see Send data.

Send broadcast data

This sample Java application shows how to send data from the local XBee device to all remote devices on the same network (broadcast) using the XBee Java Library. The application blocks during the transmission request, but you are notified if there is any error during the process.

You can locate the example in the following path:

examples/communication/SendBroadcastDataSample

Note For more information about how to get and set other parameters, see Send data to all devices of the network.

Send CoAP data (Thread devices)

This sample Java application shows how to send CoAP data to another Thread device specified by its IPv6 address.

You can find the example at the following path:
examples/communication/coap/SendCoAPDataSample

Note For more information about sending CoAP data, see Send CoAP data.

Send explicit data

This sample Java application shows how to send data in application layer (explicit) format to a remote Zigbee device on the same network as the local one using the XBee Java Library. In this example, the XBee module sends explicit data using a reliable transmission method. The application blocks during the transmission request, but you are notified if there is any error during the process.

You can locate the example in the following path:

examples/communication/explicit/SendExplicitDataSample

Note For more information about how to get and set other parameters, see Send explicit data.

Send explicit data asynchronously

This sample Java application shows how to send data in application layer (explicit) format asynchronously to a remote Zigbee device on the same network as the local one using the XBee Java Library. Transmitting data asynchronously means the execution is not blocked during the transmit request, but you cannot determine if the data was sent successfully.

You can locate the example in the following path:

examples/communication/explicit/SendExplicitDataAsyncSample

Note For more information about how to get and set other parameters, see Send explicit data.

Send broadcast explicit data

This sample Java application shows how to send data in application layer (explicit) format to all remote devices on the same network (broadcast) as the local one using the XBee Java Library. The application blocks during the transmission request, but you are notified if there is any error during the process.

You can locate the example in the following path:

examples/communication/explicit/SendBroadcastExplicitDataSample

Note For more information about how to get and set other parameters, see Send explicit data to all devices in the network.

Send IP data (IP devices)

This sample Java application shows how to send IP data to another device specified by its IP address and port number.

You can find the example at the following path:

examples/communication/ip/SendIPDataSample

Note For more information about sending IP data, see Send IP data.

Send IPv6 data (Thread devices)

This sample Java application shows how to send UDP data to another device specified by its IPv6 address and port number.

You can find the example at the following path:
examples/communication/ip/SendIPv6DataSample

Note For more information about sending IP data, see Send IPv6 data.

Send SMS (Cellular devices)

This sample Java application shows how to send an SMS to a phone or Cellular device.

You can find the example at the following path:

examples/communication/cellular/SendSMSSample

Note For more information about how to send SMS messages, see Send SMS messages.

Send UDP data (IP devices)

This sample Java application shows how to send UDP data to another device specified by its IP address and port number.

You can find the example at the following path:

examples/communication/ip/SendUDPDataSample

Note For more information about sending IP data, see Send IP data.

Receive data

This sample Java application shows how data packets are received from another XBee device on the same network.

The application prints the received data to the standard output in ASCII and hexadecimal formats after the sender address.

You can locate the example in the following path:

examples/communication/ReceiveDataSample

Note For more information about how to get and set other parameters, see Data reception callback.

Receive CoAP data (Thread devices)

This sample Java application shows how a Thread device receives CoAP data using a callback executed every time it receives new CoAP data.

You can find the example at the following path:
examples/communication/coap/ReceiveCoAPDataSample

Note For more information about how to receive IPv6 data, see Receive CoAP data.

Receive data polling

This sample Java application shows how data packets are received from another XBee device on the same network using a polling mechanism.

The application prints the data that was received to the standard output in ASCII and hexadecimal formats after the sender address.

You can locate the example in the following path:

examples/communication/ReceiveDataPollingSample

Note For more information about how to get and set other parameters, see Polling for data.

Receive explicit data

This sample Java application shows how a Zigbee device receives data in application layer (explicit) format using a callback executed every time new data is received. Before receiving data in explicit format, the API output mode of the Zigbee device is configured in explicit mode.

You can locate the example in the following path:

examples/communication/explicit/ReceiveExplicitDataSample

Note For more information about how to get and set other parameters, see Explicit data reception callback.

Receive explicit data polling

This sample Java application shows how a Zigbee device receives data in application layer (explicit) format using a polling mechanism. Before receiving data in explicit format, the API output mode of the Zigbee device is configured in explicit mode.

You can locate the example in the following path:

examples/communication/explicit/ReceiveExplicitDataPollingSample

Note For more information about how to get and set other parameters, see Polling for data.

Receive IP data (IP devices)

This sample Java application shows how an IP device receives IP data using a callback executed every time it receives new IP data.

You can find the example at the following path:

examples/communication/ip/ReceiveIPDataSample

Note For more information about how to receive IP data using the polling mechanism, see Receive IP data.

Receive IPv6 data (Thread devices)

This sample Java application shows how a Thread device receives IPv6 data using a callback executed every time it receives new IPv6 data.

You can find the example at the following path:
examples/communication/ip/ReceiveIPv6DataSample

Note For more information about how to receive IPv6 data, see Receive IPv6 data.

Receive SMS (Cellular devices)

This sample Java application shows how to receive SMS messages configuring a callback executed when new SMS is received.

You can find the example at the following path:

examples/communication/cellular/ReceiveSMSSample

Note For more information about how to receive SMS messages see Receive SMS messages.

Receive modem status

This sample Java application shows how modem status packets (events related to the device and the network) are handled using the API.

The application prints the modem status events to the standard output when received.

You can locate the example in the following path:

examples/communication/ReceiveModemStatusSample

Note For more information about how to get and set other parameters, see Receive modem status events.

Connect to echo server (IP devices)

This sample Java application shows how IP devices can connect to an echo server, send data to it and reads the echoed data.

You can find the example at the following path:

examples/communication/ip/ConnectToEchoServerSample

Note For more information about how to send and receive IP data, see Send IP data and Receive IP data.

Knock Knock (IP devices)

This sample Java application demonstrates the communication with IP devices. It starts a simple web server and connects to it by sending a message to start a Knock Knock joke.

You can find the example at the following path:

examples/communication/ip/KnockKnockSample

Note For more information about how to send and receive IP data, see Send IP data and Receive IP data.

IO samples

Local DIO

This sample Java application shows how to set and read XBee digital lines of the device attached to the serial/USB port of your PC.

The application configures two IO lines of the XBee device: one as a digital input (button) and the other as a digital output (LED). The application reads the status of the input line periodically and updates the output to follow the input.

While you press the push button, the LED should be lit.

You can locate the example in the following path:

examples/io/LocalDIOSample

Note For more information about how to get and set other parameters, see Digital input/output.

Local ADC

This sample Java application shows how to read XBee analog inputs of the device attached to the serial/USB port of your PC.

The application configures an IO line of the XBee device as ADC. It periodically reads its value and prints it in the output console.

You can locate the example in the following path:

examples/io/LocalADCSample

Note For more information about how to get and set other parameters, see ADC.

Remote DIO

This sample Java application shows how to set and read XBee digital lines of remote devices.

The application configures two IO lines of the XBee devices: one in the remote device as a digital input (button) and the other in the local device as a digital output (LED). The application reads the status of the input line periodically and updates the output to follow the input.

While you press the push button, the LED should be lit.

You can locate the example in the following path:

examples/io/RemoteDIOSample

Note For more information about how to get and set other parameters, see Digital input/output.

Remote ADC

This sample Java application shows how to read XBee analog inputs of remote XBee devices.

The application configures an IO line of the remote XBee device as ADC. It periodically reads its value and prints it in the output console.

You can locate the example in the following path:

examples/io/RemoteADCSample

Note For more information about how to get and set other parameters, see ADC.

IO sampling

This sample Java application shows how to configure a remote device to send automatic IO samples and how to read them from the local module.

The application configures two IO lines of the remote XBee device: one as digital input (button) and the other as ADC, and enables periodic sampling and change detection. The device sends a sample every five seconds containing the values of the two monitored lines. The device sends another sample every time the button is pressed or released, which only contains the value of this digital line.

The application registers a listener in the local device to receive and handle all IO samples sent by the remote XBee module.

You can locate the example in the following path:

examples/io/IOSamplingSample

Note For more information about how to get and set other parameters, see Register an IO sample listener.

XBee Java Library API reference

Welcome to the XBee Java Library API reference.

This guide provides detailed information about the features and capabilities of this product. You can find additional detailed reference information in the XBee Java Library Javadoc. The Javadoc documentation is helpful for developers who are interested in using and extending the library functionality.

The Javadoc is available in two ways:

- Off-line use. The library is included in the XBee Java Library release package available at GitHub, inside the javadoc directory.
- Online on our site. You can browse the documentation at XBee Java Library Javadoc.

Frequently Asked Questions (FAQs)

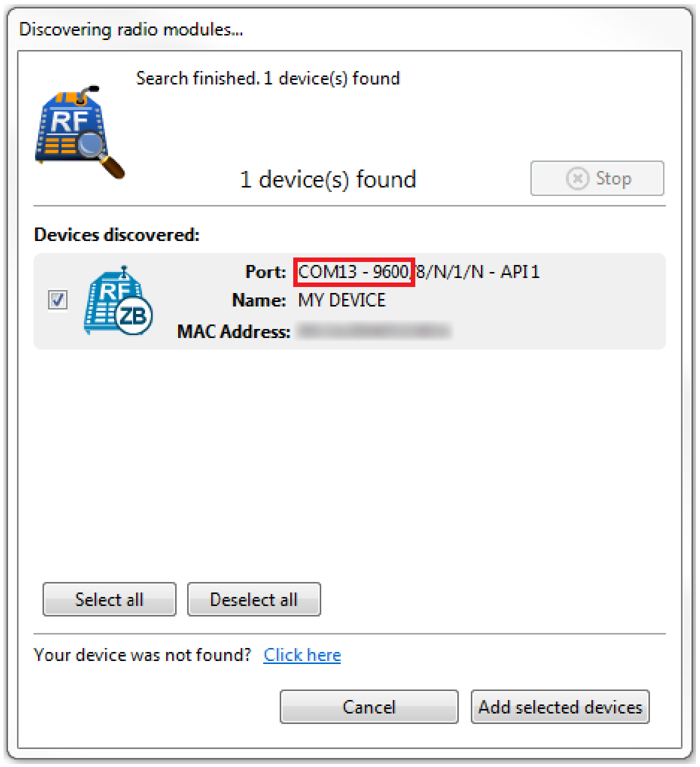
The FAQ section contains answers to general questions related to the XBee Java Library.

What is XCTU and how do I download it?

XCTU is a free multi-platform application designed to enable developers to interact with Digi RF modules through a simple-to-use graphical interface. You can download it at www.digi.com/xctu.

How do I find the serial port and baud rate of my module?

Open the XCTU application, and click Discover radio modules connected to your machine button. Select all ports to be scanned, click **Next** and then **Finish**. Once the discovery process has finished, a new window notifies you how many devices have been found and their details. The serial port and the baud rate are shown in the **Port** label.


Discovering radio modules...

Search finished. 1 device(s) found

1 device(s) found

Devices discovered:

Port: COM13 - 9600/8/N/1/N - API 1
Name: MY DEVICE
MAC Address: [redacted]

Your device was not found? Click here

Note In UNIX systems, the complete name of the serial port contains the **/dev/ prefix**.

Can I use the XBee Java Library with modules in AT operating mode?

No, the XBee Java Library only supports API and API Escaped operating modes.

Additional resources

Contribute now!

All ideas and contributions are welcome. If you find a bug or want to request new features, you can report these on GitHub.

Digi Forum

The Digi Forum is the place where you can ask questions and receive answers from other members of the community.