

# bada Tutorial: Communication (Telephony, Messaging, Network)

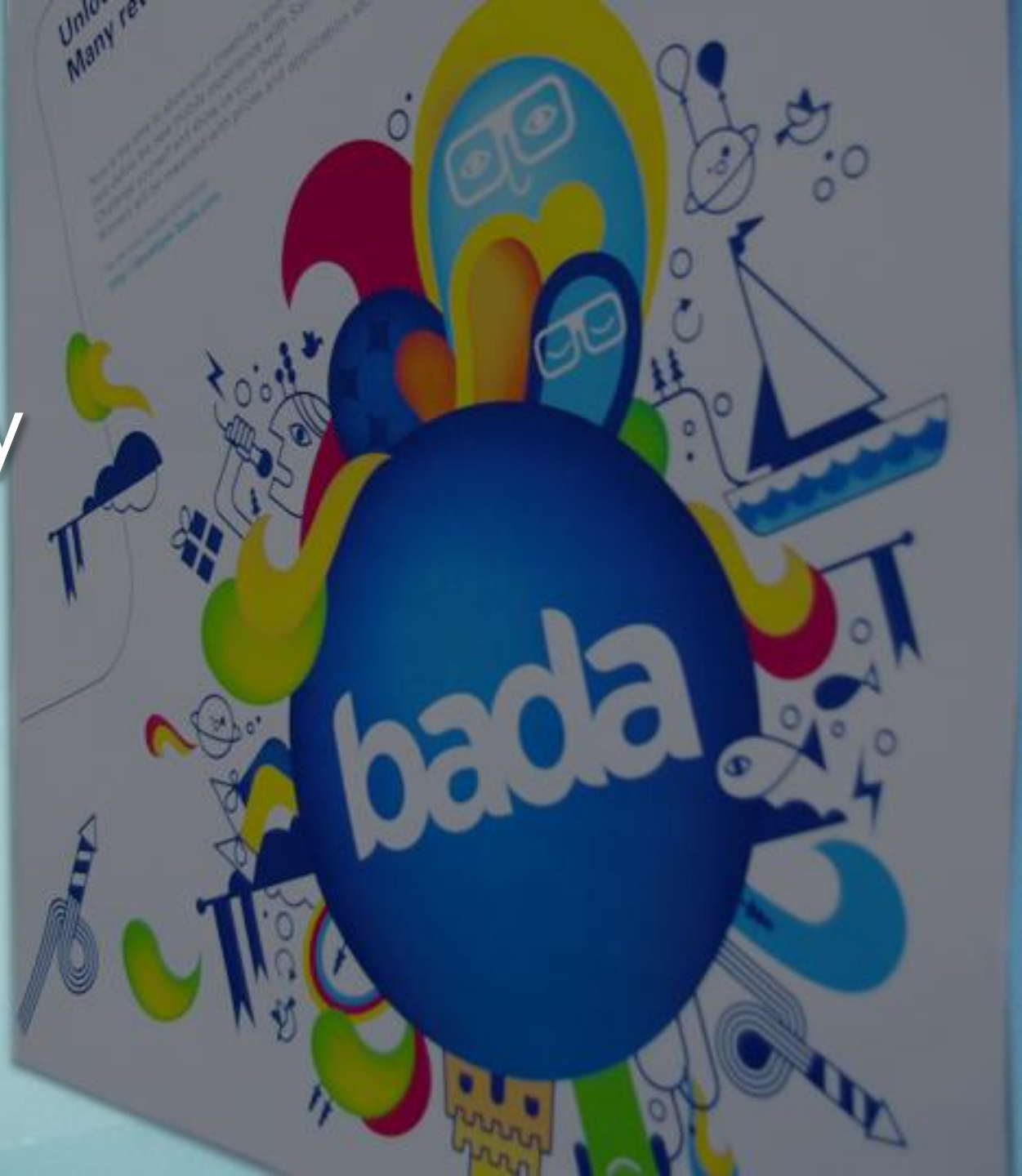


# Contents

- Telephony
- Messaging
- Network
  - Net
  - Sockets
  - Http
  - Wi-Fi
  - Bluetooth



# Telephony



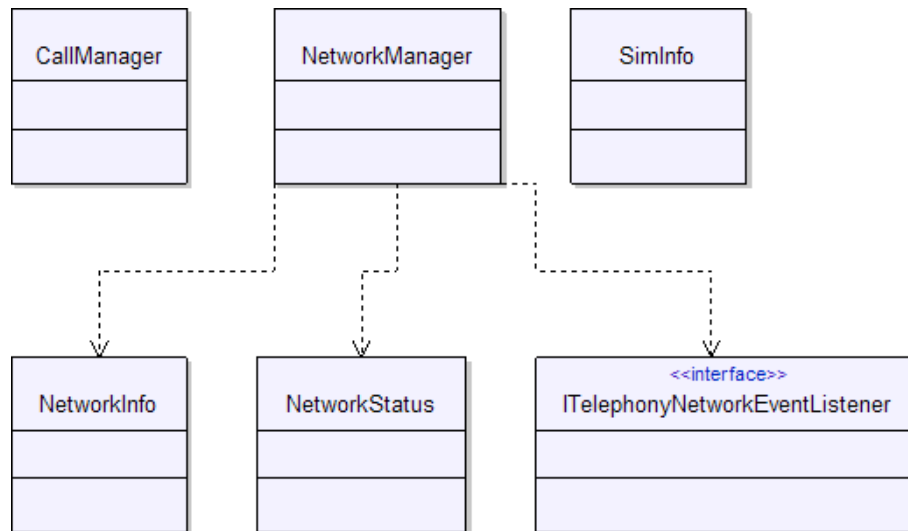
# Contents

- Essential Classes and Relationships
- Overview
- CallManager
  - Example: Get Current Call Status
- NetworkManager
  - NetworkInfo
  - Example: Get Active Network Information
  - Example: Get Status Change Notifications
- SimInfo
  - Example: Get SIM Card Information
- FAQ
- Review
- Answers



# Essential Classes and Relationships

Feature	Provided by
Provides information about the current call.	CallManager
Provides information about the current active network and its status.	NetworkManager, NetworkInfo, NetworkStatus, ITelephonyNetworkEventListener
Provides information about the currently inserted SIM card.	SimInfo



# Overview

- The Telephony namespace contains classes and interfaces to access the device's telephony capabilities.
- Key features of the telephony service are related to calls, the network and the SIM card:
  - Call:
    - Getting the current call type and call status.
  - Network:
    - Getting notification of network status changes for voice and data services.
    - Getting network information, such as PLMN, MCC, MNC, Cell ID, and LAC.
  - SIM card:
    - Getting SIM card information, such as SPN, MCC, MNC, ICC-ID, and operator name.



# CallManager

- The call manager provides information about the current call type and status using the following 2 methods:
  - `GetCurrentCallStatus()`
  - `GetCurrentCallType()`
- The call status is enumerated as shown below:

Telephony::CallStatus	Description
CALL_STATUS_IDLE	No call in progress.
CALL_STATUS_ACTIVE	A voice or video call in progress.

- The call type is enumerated as shown below:

Telephony::CallType	Description
TYPE_VOICE_CALL	The call is voice only.
TYPE_VIDEO_CALL	The call is video and voice.



# Example: Get Current Call Status

Get the status of the current call and the call type.

– Open `\<BADA_SDK_HOME>\Examples\Communication\src\Telephony\TelephonyExample.cpp, GetCallInfoExample()`

1. **Construct a CallManager:**

```
CallManager::Construct()
```

2. **Get the status of the current call:**

```
CallManager::GetCurrentCallStatus()
```

3. **If the status is active, get the type of the current call:**

```
CallManager::GetCurrentCallType()
```



# NetworkManager

- The `NetworkManager` class returns a `NetworkInfo` object and a `NetworkStatus` object.
- Pass new `NetworkInfo` and `NetworkStatus` objects into their respective methods to get current information about the network and its status.
  - `GetNetworkInfo (&networkInfo)`
  - `GetNetworkStatus (&networkStatus)`

NetworkInfo
<code>GetCellId ()</code>
<code>GetLac ()</code>
<code>GetMcc ()</code>
<code>GetMnc ()</code>
<code>GetPlmn ()</code>

NetworkStatus
<code>IsCallServiceAvailable ()</code>
<code>IsDataServiceAvailable ()</code>
<code>IsRoaming ()</code>

# NetworkInfo

The `NetworkManager`'s `GetNetworkInfo()` method returns a `NetworkInfo` object that contains information about the network that the device is currently connected to or registered with.

Method	Description
<code>GetCellId()</code>	The cell ID of the mobile device.
<code>GetLac()</code>	The Location Area Code.
<code>GetMcc()</code>	The Mobile Country Code.
<code>GetMnc()</code>	The Mobile Network Code.
<code>GetPlmn()</code>	The Public Land Mobile Network.

# Example: Get Active Network Information (1/2)

Get information on the active network and its status.

– Open `\<BADA_SDK_HOME>\Examples\Communication\src\Telephony\TelephonyExample.cpp, GetNetworkInfoExample()`

1. **Construct a NetworkManager:**

```
NetworkManager::Construct()
```

2. **Get the instance of NetworkStatus:**

```
NetworkManager::GetNetworkStatus(networkStatus)
```

3. **Get the network status using getter methods:**

```
NetworkStatus::IsCallServiceAvailable()
```

```
NetworkStatus::IsDataServiceAvailable()
```

```
NetworkStatus::IsRoaming()
```

4. **Get a NetworkInfo instance:**

```
NetworkManager::GetNetworkInfo()
```



# Example: Get Active Network Information (2/2)

## 5. Get network information:

```
NetworkInfo::GetMnc()
```

```
NetworkInfo::GetMcc()
```

```
NetworkInfo::GetCellId()
```

```
NetworkInfo::GetLac()
```

```
NetworkInfo::GetPlmn()
```



# Example: Get Status Change Notifications

Get notifications when the network status changes.

- Open `\<BADA_SDK_HOME>\Examples\Communication\src\Telephony\TelephonyExample.cpp`,  
`GetNetworkNotificationExample()`

1. **Implement an `ITelephonyNetworkEventListener` interface listener and create an instance.**
2. **Construct a `NetworkManager` with the listener:**  
`NetworkManager::Construct(listener)`
3. **When network status changes, the `OnTelephonyNetworkStatusChanged()` event handler fires:**  
`ITelephonyNetworkEventListener::OnTelephonyNetworkStatusChanged()`
4. **Get the status changes from the `NetworkStatus` object:**  
`NetworkStatus::IsCallServiceAvailable()`  
`NetworkStatus::IsDataServiceAvailable()`  
`NetworkStatus::IsRoaming()`



# SimInfo

`SimInfo` contains information about the current SIM card.

Method	Description
<code>GetIccId()</code>	Returns the Integrated Circuit Card ID (ICC-ID).
<code>GetMcc()</code>	Returns the Mobile Country Code of the SIM IMSI (International Mobile Subscriber Identity).
<code>GetMnc()</code>	Returns the Mobile Network Code of the SIM IMSI.
<code>GetOperatorName()</code>	Returns the mobile network operator name.
<code>GetSpn()</code>	Returns the service provider name.
<code>IsAvailable()</code>	Checks if the SIM card is available in the device or not.

# Example: Get SIM Card Information

Get SIM card information using a `SimInfo` object.

- Open `\<BADA_SDK_HOME>\Examples\Communication\src\Telephony\TelephonyExample.cpp, GetSimInfoExample()`

1. **Construct a `SimInfo`:**

```
SimInfo::Construct()
```

2. **Get SIM card information using `SimInfo` methods:**

```
GetMnc()
```

```
GetMcc()
```

```
GetSpn()
```

```
GetIccId()
```

```
GetOperatorName()
```

```
IsAvailable()
```



# FAQ (1/2)

- Why does SPN return an empty string?
  - SPN is not a mandatory field. Some of the SIM cards that do not have it return an empty string.
- What is the ICC-ID?
  - ICC-ID is an acronym for Integrated Circuit Card ID.
  - The ID uniquely identifies each SIM internationally.
  - ICC-ID is stored on a SIM card and printed or engraved on it.
  - ITU-T recommendation E.118 defines ICC-ID.
  - They are 18- or 19-digit numbers.



# FAQ (2/2)

## What are the MCC and MNC?

- MCC (Mobile Country Code) is a 3-digit code that identifies a country. More than one MCC can be assigned to a country.
- MNC (Mobile Network Code) is a 2 or 3-digit code that uniquely identifies mobile phone operators.
- MCC and MNC are defined in ITU E.212 (“Land Mobile Numbering Plan”).
- The following are MCC and MNC examples:

Carrier name (country)	MCC	MNC
British Telecom (UK)	234	00
O2 (UK)	234	02, 10, 11
Vodafone (UK)	234	15
T-Mobile (UK)	234	30
T-Mobile (Germany)	262	01, 06
SK Telecom (Korea)	450	03, 05

# Review

1. What types of calls are there?
2. What does the `GetIccId()` method return and what does it represent?
3. Two classes have some similar methods. What are the classes? Name at least 1 similar method.
4. Which of the following are used when uniquely identifying a mobile phone internationally?
  - ICC-ID
  - MCC
  - MNC
  - SPN

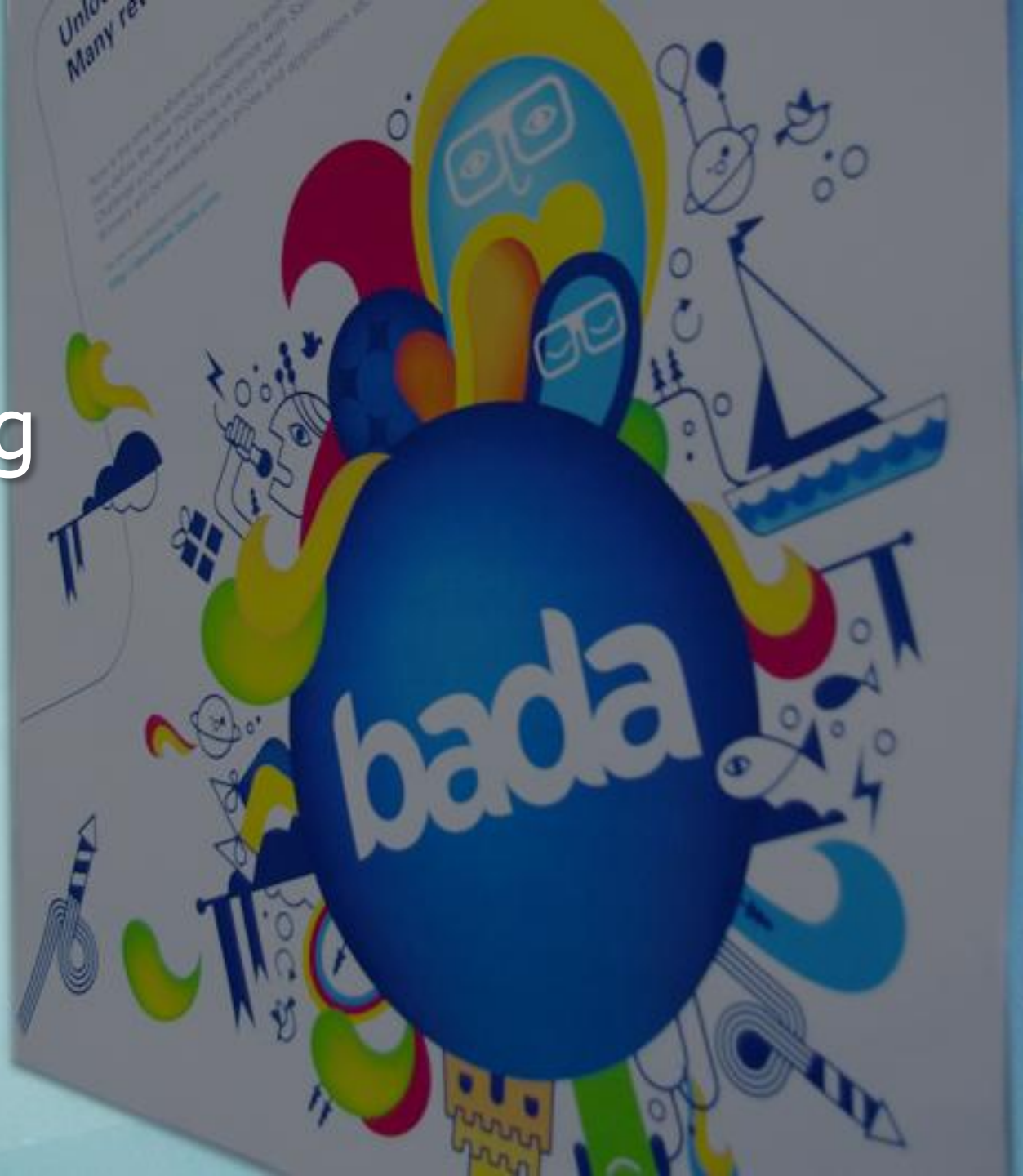


# Answers

1. Voice and video.
2. Integrated Circuit Card ID. It uniquely identifies a SIM card internationally.
3. `NetworkInfo` and `SimInfo`. Similar methods include `GetMcc()` and `GetMnc()`.
4. ICC-ID is the identity value itself.



# Messaging



# Contents

- Essential Classes
- Relationships between Classes: SMS
- Relationships between Classes: MMS
- Relationships between Classes: Email
- Overview
- Manager Classes
- Data Classes
  - RecipientList
- Sending Messages Programmatically
  - Example: Send an SMS Programmatically
  - Example: Send an MMS Programmatically
  - Example: Send an Email Programmatically
- Sample Application: "MessageSender"
- Event Injector
- FAQ
- Review
- Answers

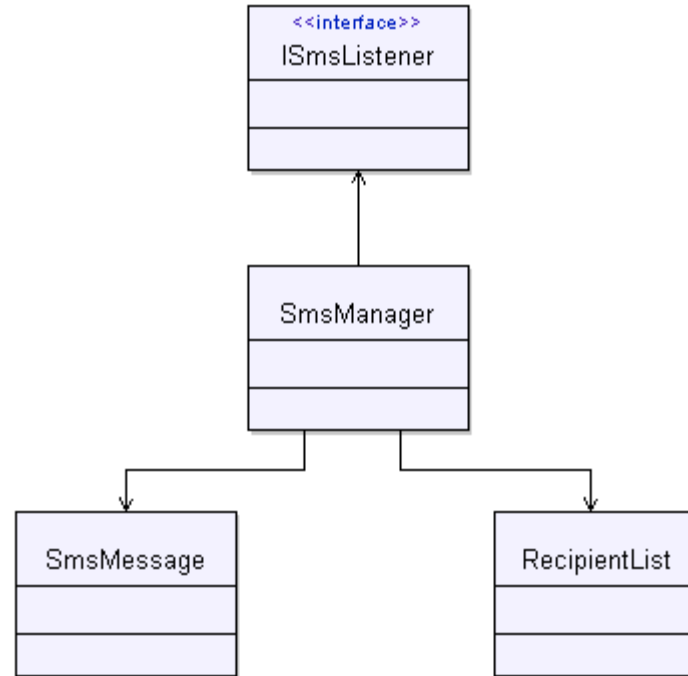


# Essential Classes

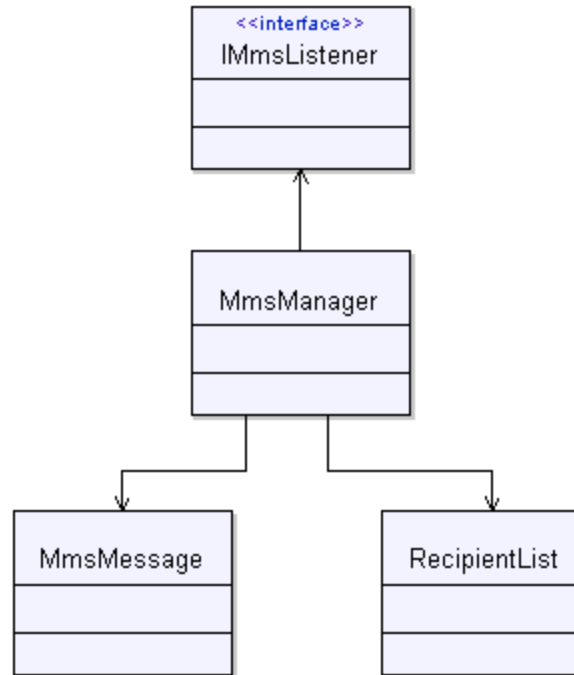
Feature	Provided by
Provides an SMS message containing body text.	SmsMessage
Provides an MMS message containing a subject, body text, and attachments.	MmsMessage
Provides an email message containing a subject, body text, and attachments.	EmailMessage
Sends SMS messages.	SmsManager
Sends MMS messages with attachments.	MmsManager
Sends email messages with attachments.	EmailManager
Provides an asynchronous listener for sending SMS messages.	ISmsListener
Provides an asynchronous listener for sending MMS messages.	IMmsListener
Provides an asynchronous listener for sending email messages.	IEmailListener



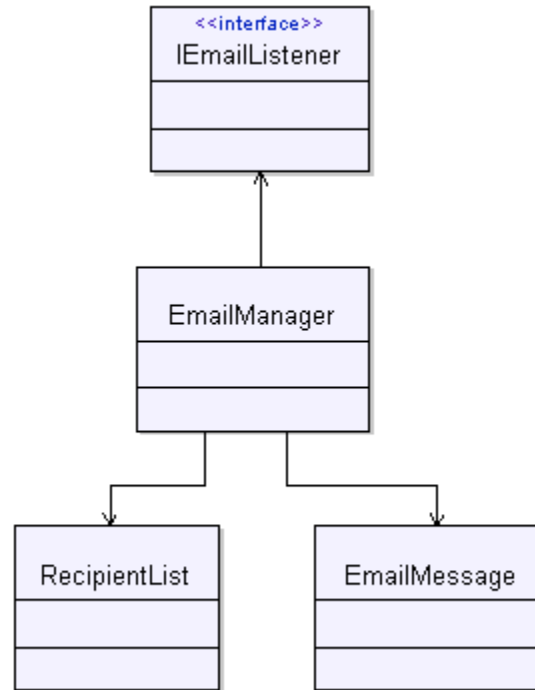
# Relationships between Classes: SMS



# Relationships between Classes: MMS



# Relationships between Classes: Email



# Overview

- The Messaging namespace contains classes and interfaces that provide SMS, MMS, and email features.
- Key features include:
  - Creating and sending SMS messages.
  - Creating and sending MMS and email messages with attachments.



# Manager Classes

- SmsManager
  - Adds listener to receive asynchronous results.
  - Sends SMS messages.
- MmsManager
  - Adds listener to receive asynchronous results.
  - Sends MMS messages with attachments (image, video and audio).
- EmailManager
  - Adds listener to receive asynchronous results.
  - Sends email messages with attachments.



# Data Classes

- `SmsMessage`
  - The abstract data type class for SMS messages.
  - Creates the body text.
- `MmsMessage`
  - The abstract data type class for MMS messages.
  - Creates the body text, subject, and attachments.
- `EmailMessage`
  - The abstract data type class for email messages.
  - Creates the body text, subject, and attachments.
- `RecipientList`
  - The abstract data type class for recipients.
  - Adds, removes, sets, and gets recipients.



# RecipientList

- The `RecipientList` contains a list of phone numbers or addresses to send messages to.
- A recipient list has 3 distinct sections to it, divided using the `RecipientType` enumeration.

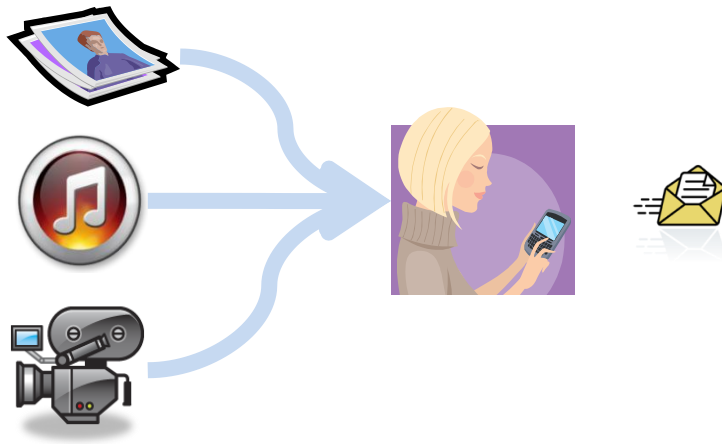
Type	Description	Used with
TO	The list of people the message is addressed to.	SMS, MMS, and email
CC	The list of people the message is carbon copied to.	MMS and email
BCC	The list of people the message is blind carbon copied to.	MMS and email

- You can create a recipient list using a particular recipient list type, and add another list of another type.
- You can also add or remove individual recipients.
- When getting a list of recipients, you must specify what type they are, TO, CC, or BCC.

# Sending Messages Programmatically (1/3)

You can programmatically send SMS, MMS, and email messages using below classes:

- SmsManager, SmsMessage
- MmsManager, MmsMessage
- EmailManager, EmailMessage



# Sending Messages Programmatically (2/3)

- You can send all message types to multiple recipients.
- You can add attachments to MMS and email messages:
  - Each attachment has size limitations. For more information, see the API Reference.
  - Email attachments: The file format does not matter.
  - MMS attachments: A subset of the MMS specification is supported.
    - Image only
    - Audio only
    - Video only
    - Image and audio clip



# Sending Messages Programmatically (3/3)

## Status report:

- After your application sends an SMS, MMS, or email message, it waits for the asynchronous sent status message.
- When sending an MMS or email message, you receive one status report regardless of the number of recipients.
- When sending an SMS message, you receive a separate status report for each recipient.



# Example: Send an SMS Programmatically

## Send an SMS message.

- Open `<BADA_SDK_HOME>\Examples\Communication\src\Messaging\SendSmsMessage.cpp`

1. Implement an `ISmsListener` and create an instance.

2. Construct an `SmsManager`:

```
SmsManager::Construct(listener)
```

3. Set the body text:

```
SmsMessage::SetText(message)
```

4. Create a recipient list and add recipients:

```
RecipientList::Add(type, recipient)
```

5. Send the message with recipients:

```
SmsManager::Send(smsMessage, recipientList, true)
```



# Example: Send an MMS Programmatically

## Send an MMS message.

– Open `<BADA_SDK_HOME>\Examples\Communication\src\Messaging\SendMmsMessage.cpp`

1. **Implement an `IMmsListener` and create an instance.**

2. **Construct an `MmsManager`:**

```
MmsManager::Construct(listener)
```

3. **Set the body text:**

```
MmsMessage::SetText(message)
```

4. **Set the subject:**

```
MmsMessage::SetSubject(subject)
```

5. **Create and add recipients:**

```
RecipientList::Add(type, recipient)
```

6. **Add attachments:**

```
MmsMessage::AddAttachment(format, filePath)
```

7. **Send the message:**

```
MmsManager::Send(mmsMessage, recipientList, true)
```

# Example: Send an Email Programmatically

## Send an email message.

– Open `<BADA_SDK_HOME>\Examples\Communication\src\Messaging\SendEmailMessage.cpp`

1. **Implement an `IEmailListener` and create an instance.**

2. **Construct an `EmailManager`:**

```
EmailManager::Construct(listener)
```

3. **Set the body text:**

```
EmailMessage::SetText(message)
```

4. **Set the subject:**

```
EmailMessage::SetSubject(subject)
```

5. **Create and add recipients:**

```
RecipientList::Add(type, recipient)
```

6. **Add attachments:**

```
EmailMessage::AddAttachment(filePath)
```

7. **Send the message:**

```
EmailManager::Send(emailMessage, recipientList, true)
```

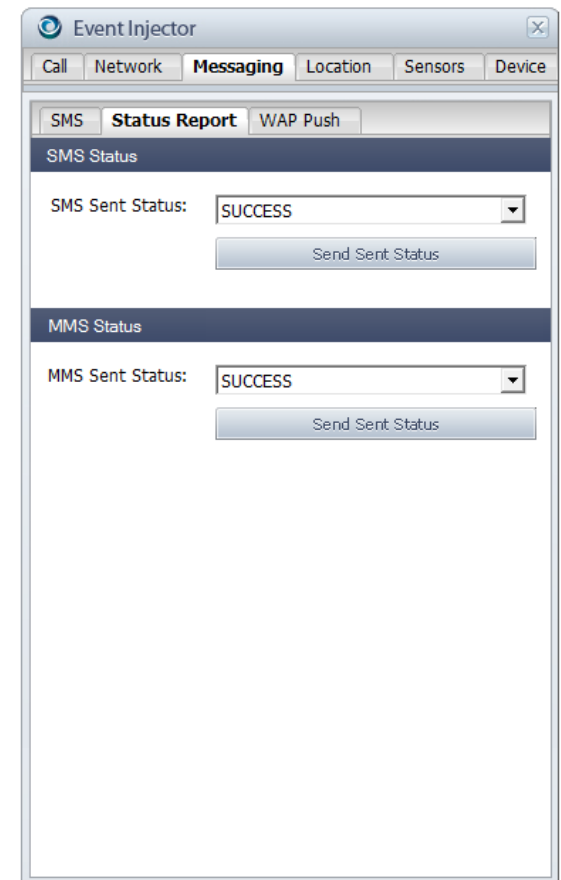
# Sample Application: "MessageSender"

- Open `\<BADA_SDK_HOME>\Samples\MessageSender\src`.
- The MessageSender application shows how to:
  - Create and send SMS messages.
  - Create and send MMS and email messages with attachments.



# Event Injector

- After your application sends an SMS or MMS message, it waits for the asynchronous sent status message.
- You can generate and issue an SMS or MMS sent status message in the **Messaging > Status Report** tab of the Event Injector.
- You can choose an SMS or MMS sent status:
  - SUCCESS
  - Other errors



Note: In case of SMS status, you must send the SMS Sent Status once for each recipient.

# FAQ

- Can I receive or handle received messages?
  - No. All message receiving is handled by the native messaging application on the device.  
Due to performance and privacy issues, bada applications do not have access to the Inbox.
- How can I use `AppControl` to send different kinds of messages?
  - See the Application tutorial.
- When I try to send a message, an `E_STORAGE_FULL` exception occurs. What do I need to do?
  - You must delete the messages in the Sentbox.
- What is the difference between the Sentbox and Outbox?
  - If a user sends a message successfully, it is saved in the Sentbox.  
If sending the message fails or gets cancelled, the message is saved in the Outbox.
- Even though I delete some messages in the Sentbox, I still get the `E_STORAGE_FULL` exception when I try to send an email.
  - In case of email messages, the deleted messages go to the Trash folder. You must delete the messages in Trash manually.

# Review

1. What different kinds of messages can you send using bada?
2. How can you send a message to more than one person?
3. Which message formats support attachments, and what method is used to add attachments?
4. Can you send file attachments in an MMS?

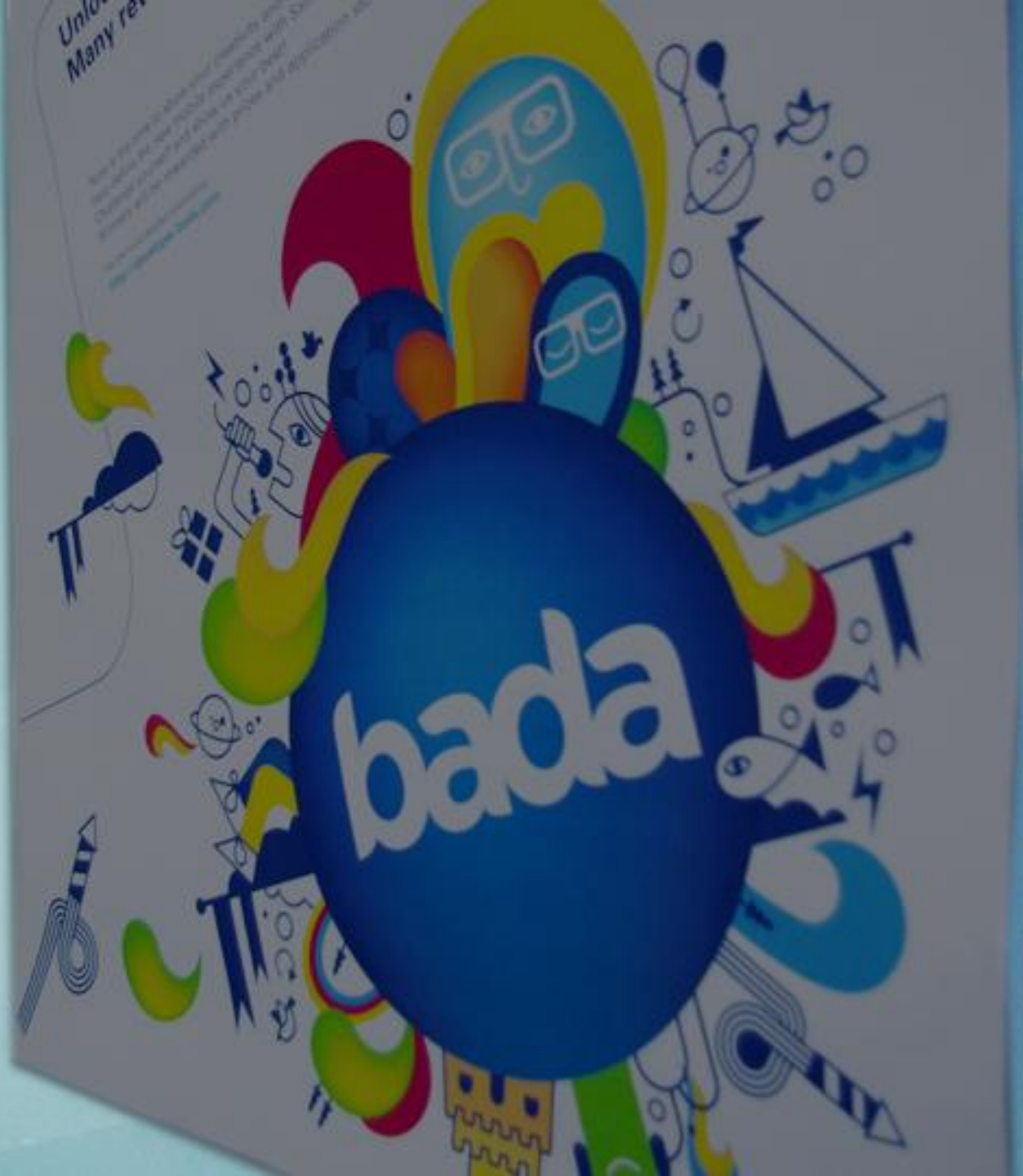


# Answers

1. SMS, MMS, and email messages.
2. Use a `RecipientList` with the `SetRecipientList()` method to send to a list of people.
3. MMS and email messages support adding attachments through the `AddAttachment()` method. The `AddAttachment()` signatures for MMS and email messages are different.
4. No. You can only send image, audio, and video attachments. Other file formats are not supported.



# Network



# Contents

- Overview
- Sync vs. Async : The Problem
- Sync vs. Async : The Solution
- Listener Interfaces



# Overview (1/2)

The Net namespace and its sub-namespaces provide a robust set of low- to high-level networking tools:

- Account management
- Connection management
- DNS and IP address management
- HTTP management
- Socket management
- Wi-Fi management
- Bluetooth management



# Overview (2/2)

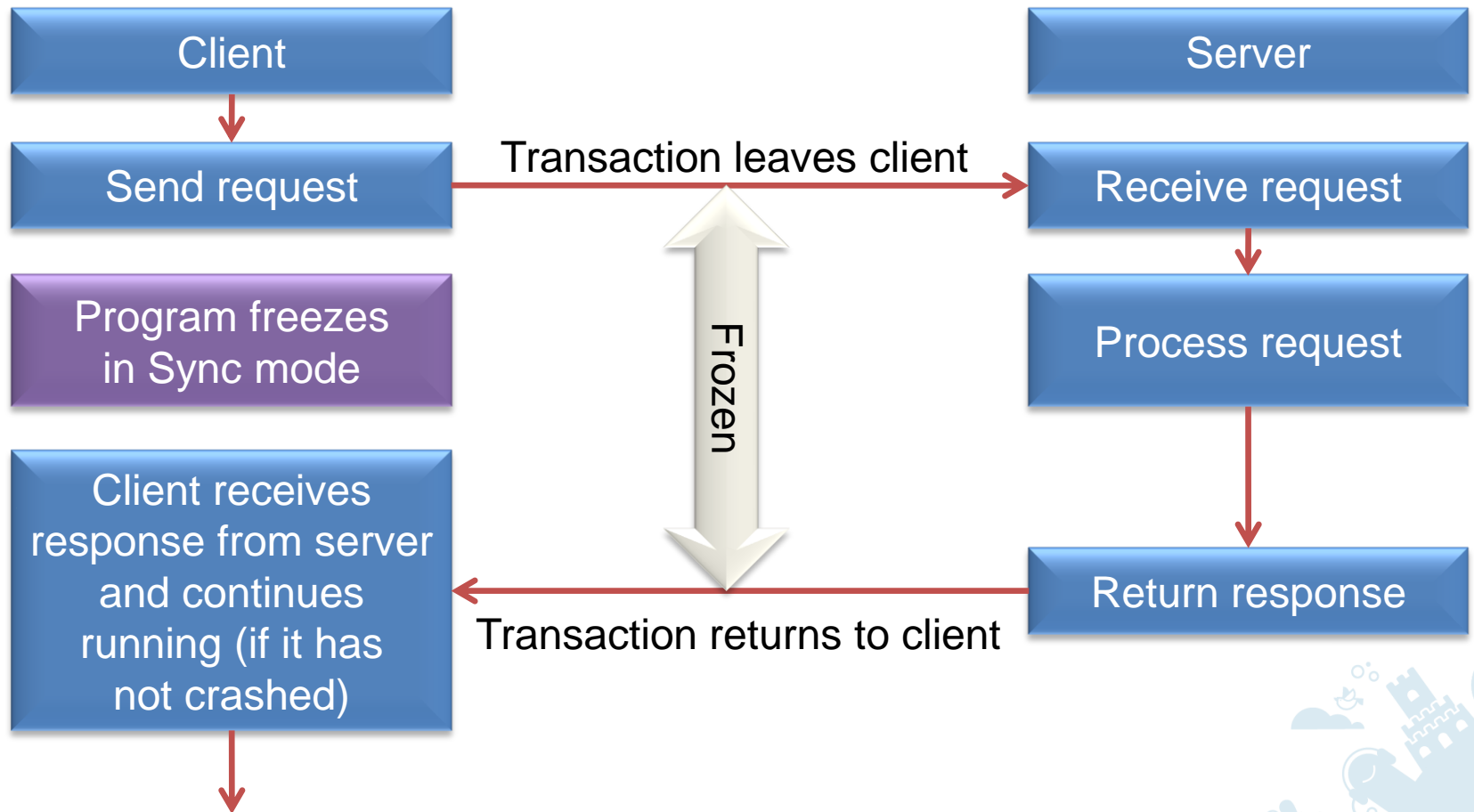
## Practical considerations:

- Classes and namespaces in the Net namespace, such as Http, and Sockets, are usually used in non-blocking mode.
- Interfaces are defined to handle events, and must be implemented to receive notifications.
  - For example, the `Osp::Net::Sockets::ISocketEventListener` event handler needs to be implemented for a socket.



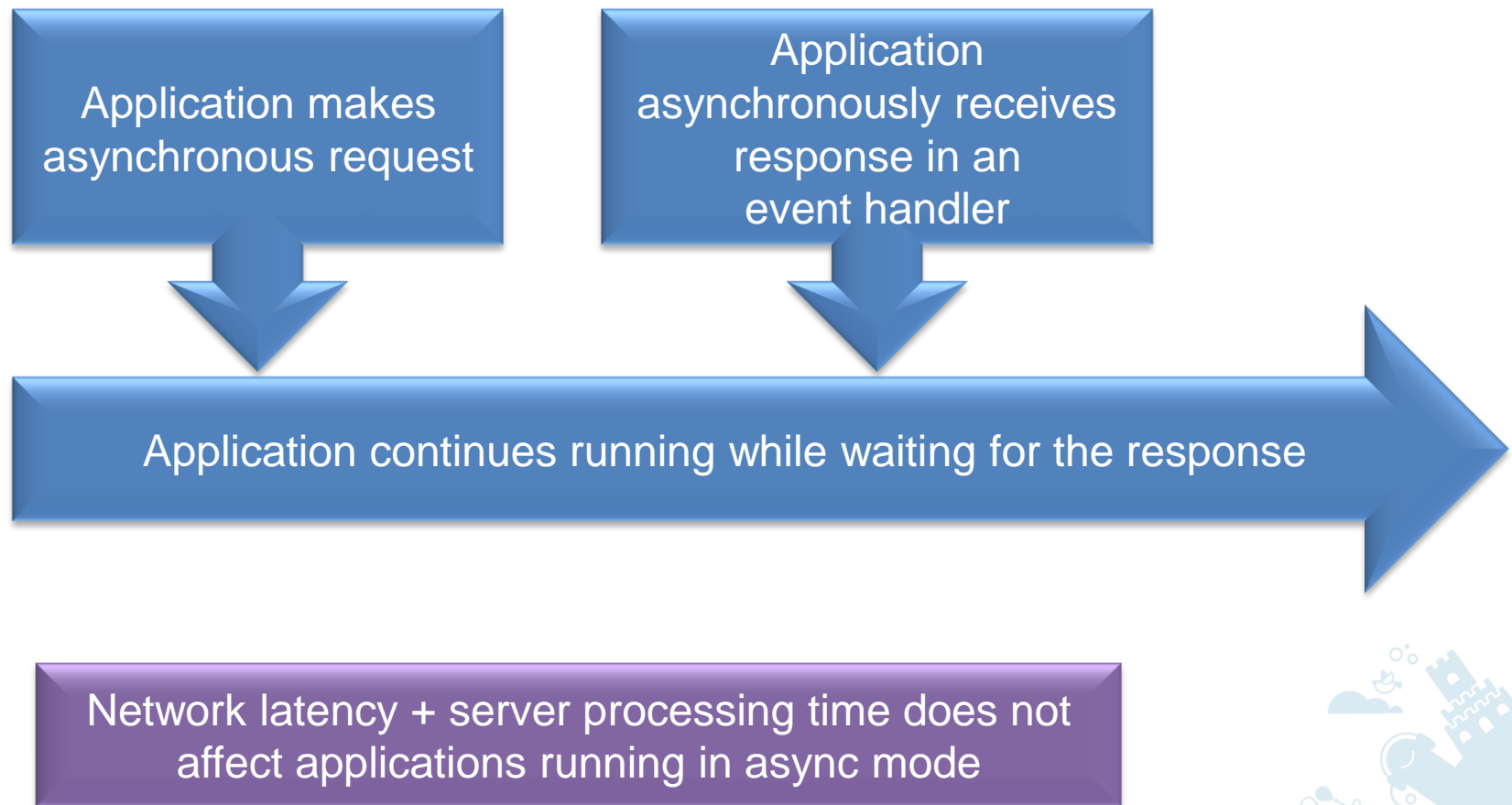
# Sync vs. Async : The Problem

Applications that use Sync mode can freeze or become non-responsive during the time the transaction is controlled by the server.



# Sync vs. Async : The Solution

The solution to the problem is to run async mode. This requires an event handler to receive a notification when the server responds to the request.

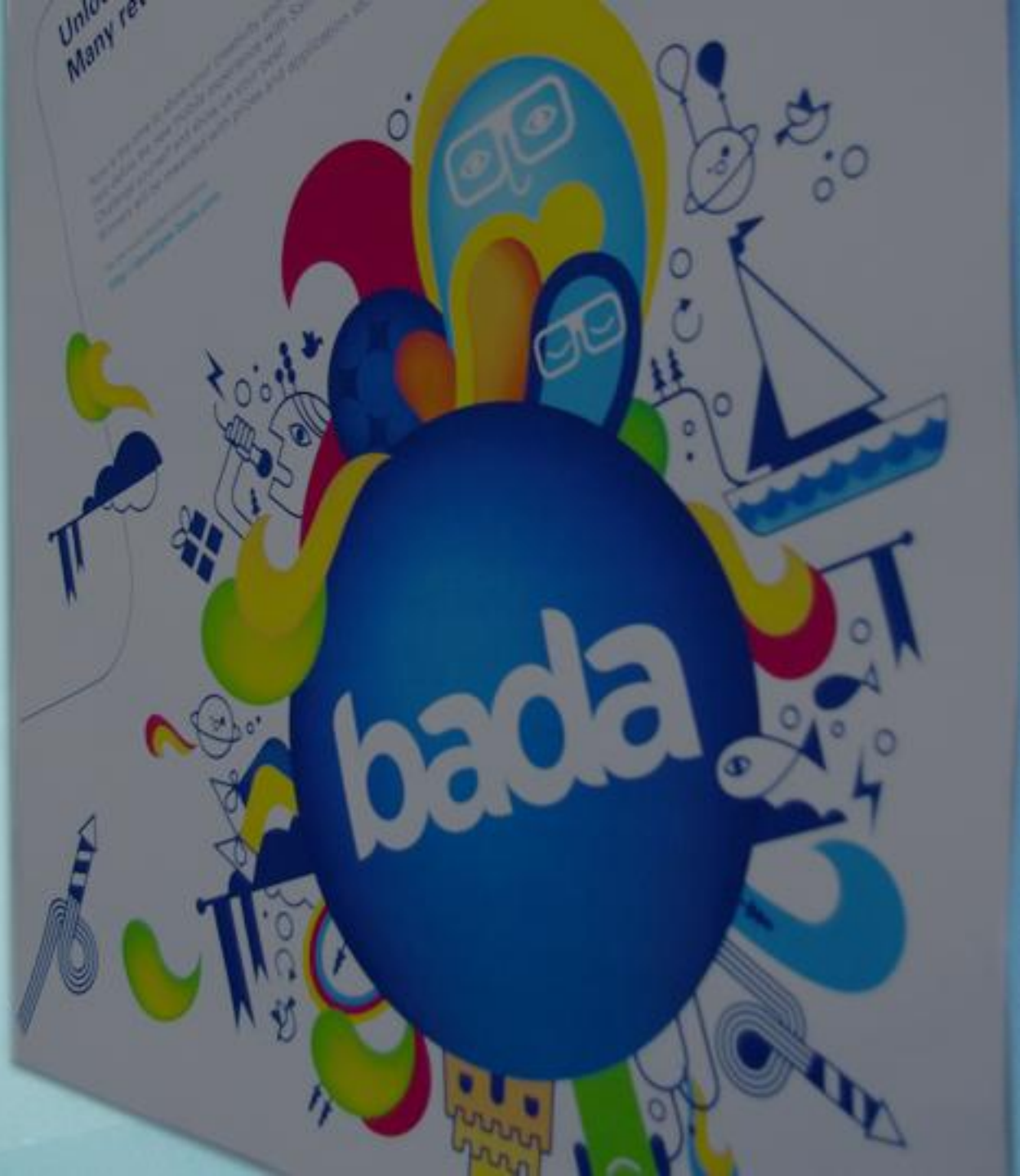


# Listener Interfaces

You need to manually implement listener interfaces to handle events and notifications. The following is a boilerplate for the `ISocketEventListener`. You can implement other listeners the same way.

```
using namespace Osp::Net::Sockets;
using namespace Osp::Base;
class TestListener :
    public Object,
    public virtual ISocketEventListener
{
    public:
        TestListener() {}
        ~TestListener() {}
        void OnSocketConnected(Socket& socket) { }
        void OnSocketClosed(Socket& socket, NetSocketClosedReason reason) { }
        void OnSocketReadyToReceive(Socket& socket) { }
        void OnSocketReadyToSend(Socket& socket) { }
        void OnSocketAccept(Socket& socket) { }
};
```

Net



# Contents

- Essential Classes
- Relationships between Classes
- Overview
- Accounts and Connections
  - Network Accounts
    - NetAccountInfo
  - Network Connections
    - Default and Custom Network Connections
    - Example: Set Preferred Network
    - Example: Control Custom Network Connection
  - More Classes and Info
- DNS
  - Example: Query DNS
- Proxy Address Setting
- FAQ
- Review
- Answers

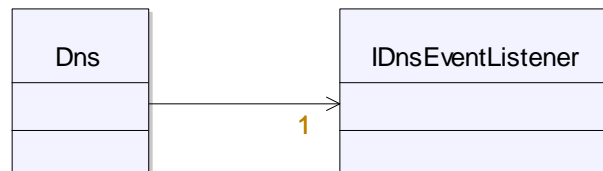
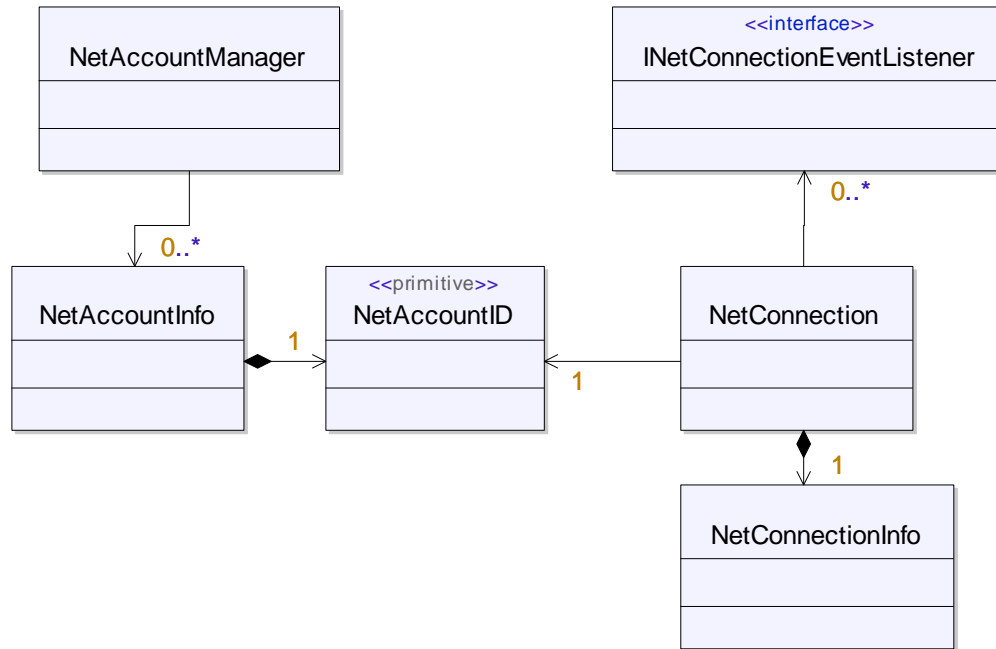


# Essential Classes

Feature	Provided by
Provides functions for creating, deleting, and administering accounts.	NetAccountManager
Encapsulates account information.	NetAccountInfo
Provides methods to create and manage network connections for data communications.	NetConnection INetConnectionEventListener
Encapsulates connection information.	NetConnectionInfo
Provides simple domain name resolution functionality.	Dns IDnsEventListener
Provides networking utilities.	NetEndPoint IPAddress (IP4Address and IP6Address)



# Relationships between Classes



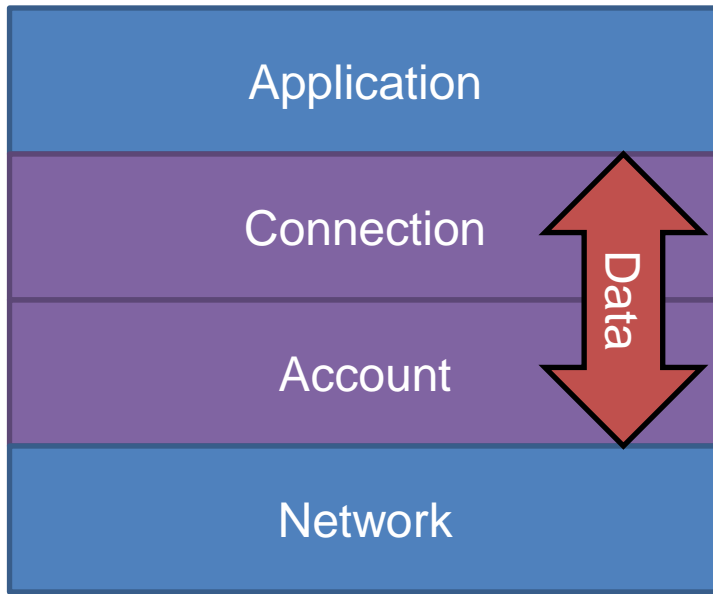
# Overview

- The Net namespace lets you work with a range of low- to high-level networking classes for data communications.
- Key features include:
  - Network account
  - Network connection



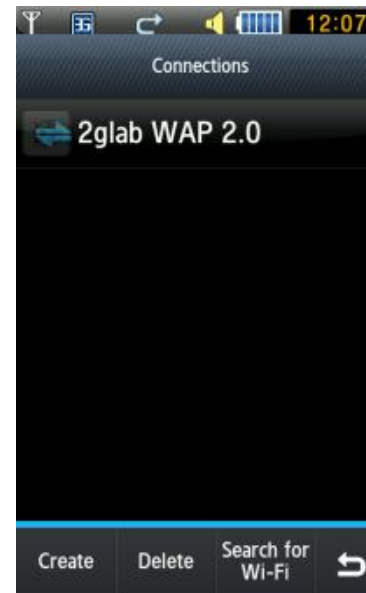
# Accounts and Connections

- Accounts are only used for configuration, while connections represent the actual run-time session. So, connections are always account-based.
- This means that connections always require a network account to access resources on the network.



# Network Accounts

- In order to access a remote network, many configuration parameters are necessary, such as the network type, access point name, protocol type, and HTTP proxy address. A network account is a set of these parameters.
- With a network account, you can:
  - Get a list of account information on the device.
  - Add a new account.
  - Delete an existing account.
  - Modify account information.



# NetAccountInfo

`NetAccountInfo` contains all the information needed to make a connection. Some properties are set together in a single method.

Get	Set	Item	Description
✔	✔	Access point name	The access point name.
✔	✔	Account ID	The account ID.
✔	✔	Account name	The account name.
✔	✔	Authentication info	The authorization type (PAP or CHAP), account ID, and account password.
✔	✔	DNS address scheme	The <code>NetAddressScheme</code> for the DNS servers.
✔	✔	Local address	The IP address.
✔	✔	Local address scheme	The <code>NetAddressScheme</code> for the account. Dynamic or static.
✔	✘	Account name max. length	The maximum length for an account name.
✔	✘	ID max. length	The maximum length for an account ID.
✔	✘	Password max. length	The maximum length for a password.
✔	✘	Operation mode	The operation mode.
✔	✘	Primary DNS address	The IP address of the primary DNS server.
✔	✔	Protocol type	The protocol type.
✔	✔	Proxy address	The IP address of the proxy, if any.
✔	✘	Secondary DNS address	The IP address of the secondary DNS server.
✔	✔	Account password	The account password.

# Network Connections

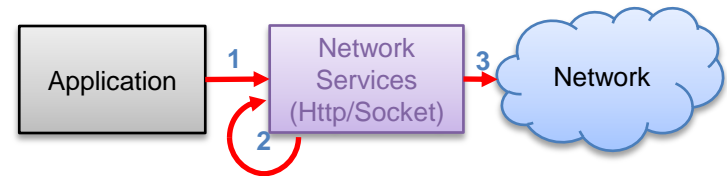
- A network connection is a runtime session for a network service. It requires a network account to establish the session. In order to access a network, a network connection must be established.
- With a network connection, you can:
  - Create a network connection using a specific network account.
  - Start a network connection.
  - Stop a network connection.
  - Register an event listener for network events.



# Default and Custom Network Connections

- Default network connection:
  - Network connections are necessary to use network-based services, but they are difficult to use. The default network connection simplifies network usage by letting the platform manage the network connection. To use the default network connection, simply do not specify a network connection and the platform uses the default internally.

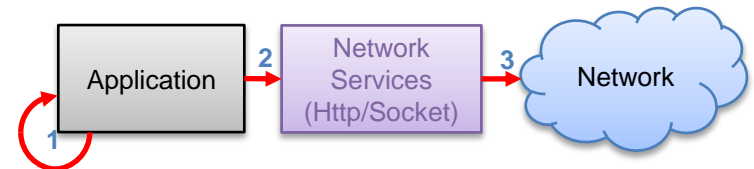
(a) A workflow using a default network connection



1. An application requests network services without a network connection.
2. Network services gets and starts a default network connection internally.
3. Network services accesses a remote network.

- Custom network connection (for advanced developers):
  - You can create a customized network connection for direct control. With a custom network connection, you can use a specific network account and start or stop the network connection at any time.

(b) A workflow using a custom network connection



1. An application creates and starts a custom network connection.
2. The application requests network services with a network connection.
3. Network services accesses a remote network.

# Example: Set Preferred Network

Set a preferred network for an application.

- Open `<BADA_SDK_HOME>\Examples\Communication\src\Net\NetPreferenceExample.cpp`

1. **Construct a NetAccountManager:**

```
Net::NetAccountManager::Construct()
```

2. **Set the preferred network for the default network connection:**

```
Net::NetAccountManager::SetNetPreference  
(NET_PS_ONLY)
```

3. **Construct a Socket or Http object:**

```
Net::Sockets::Socket::Construct()
```

```
Net::Http::HttpSession::Construct()
```

Note 1: If a developer does not need to set preferred network, skip Steps 1 and 2.

Note 2: If an application does not set a preferred network, according to system network policy, network is automatically selected.

# Example: Control Custom Network Connection

Start and stop a custom network connection.

- Open `<BADA_SDK_HOME>\Examples\Communication\src\Net\NetConnectionExample.cpp`
  
- 1. **Implement an `INetConnectionEventListener` and create an instance.**
- 2. **Get an account:**  
`NetAccountManager::GetNetAccountIdsN()`
- 3. **Construct a `NetConnection`:**  
`NetConnection::Construct(accountId)`
- 4. **Add the listener:**  
`NetConnection::AddNetConnectionListener (Listener)`
- 5. **Start the connection:**  
`NetConnection::Start()`
- 6. **Do something useful in the 'started' event handler.**
- 7. **Stop the connection:**  
`NetConnection::Stop()`
- 8. **Do something useful in the 'stopped' event handler.**



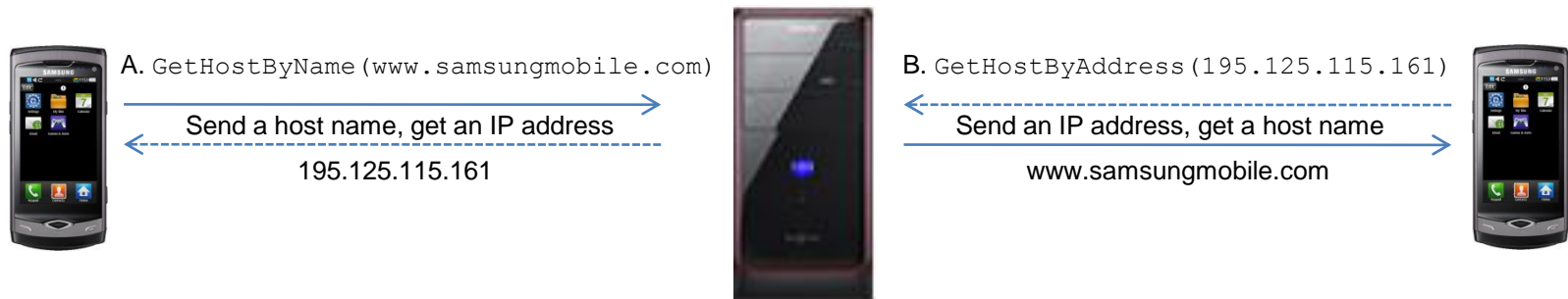
# More Classes and Info

- `NetConnectionInfo` class:
  - Contains all connection information, such as the protocol type, access point name, local address, and DNS address.
- `NetAccountId` Typedef as opposed to `NetAccountInfo` class:
  - `NetAccountId` identifies an account.
  - `NetAccountInfo` contains all account information.
- `NetStatistics` class:
  - Provides statistics on network conditions.
    - Gets the size of last sent or received data in bytes.
    - Gets the size of total sent or received data in bytes.
    - Statistics can be reset.



# DNS

- People remember friendly host names, not address octets.  
Which of these would you remember?  
SamsungMobile.com or 195.125.115.161?
- DNS translates host names into IP addresses and vice versa.
  - A. Look up IP addresses from user-friendly host names.
  - B. Look up host names from IP addresses.



# Example: Query DNS

Get an IP address from a host name.

– Open `\<BADA_SDK_HOME>\Examples\Communication\src\Net\DnsExample.cpp`, `DnsRequest()` and `OnDnsResolutionCompletedN()`

1. Request an IP address by host name:

```
Dns::GetHostByName(hostName)
```

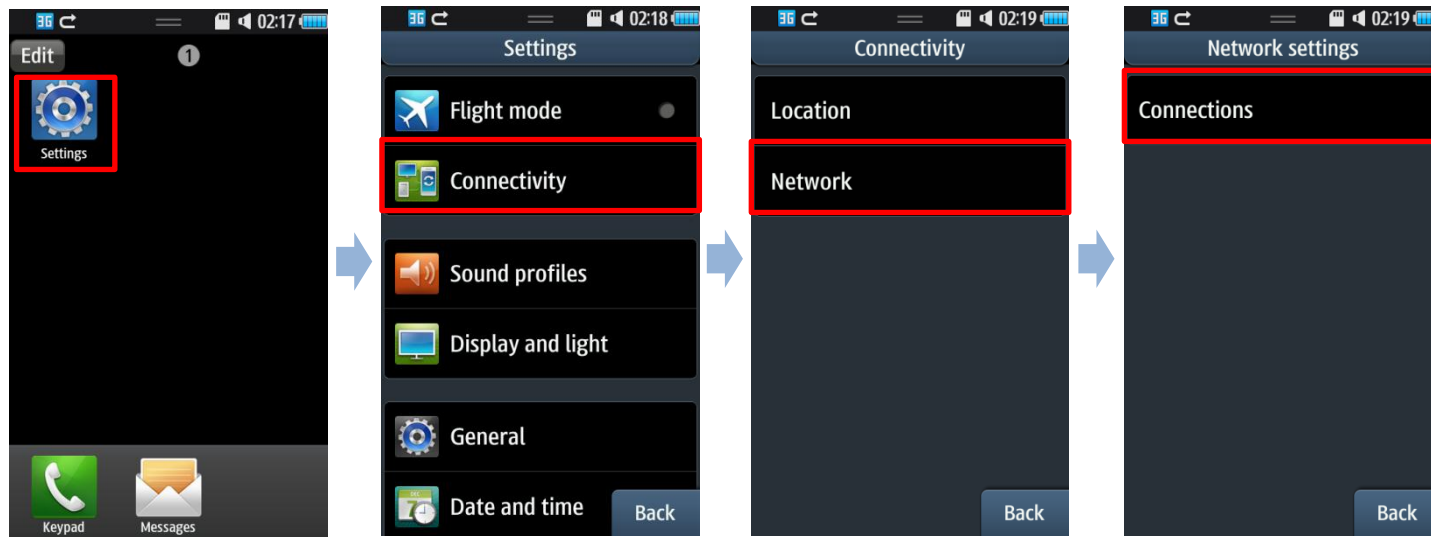
2. Retrieve the IP address from the response in the event handler:

```
IDnsEventListener::OnDnsResolutionCompletedN  
(ipHostEntry, r)
```

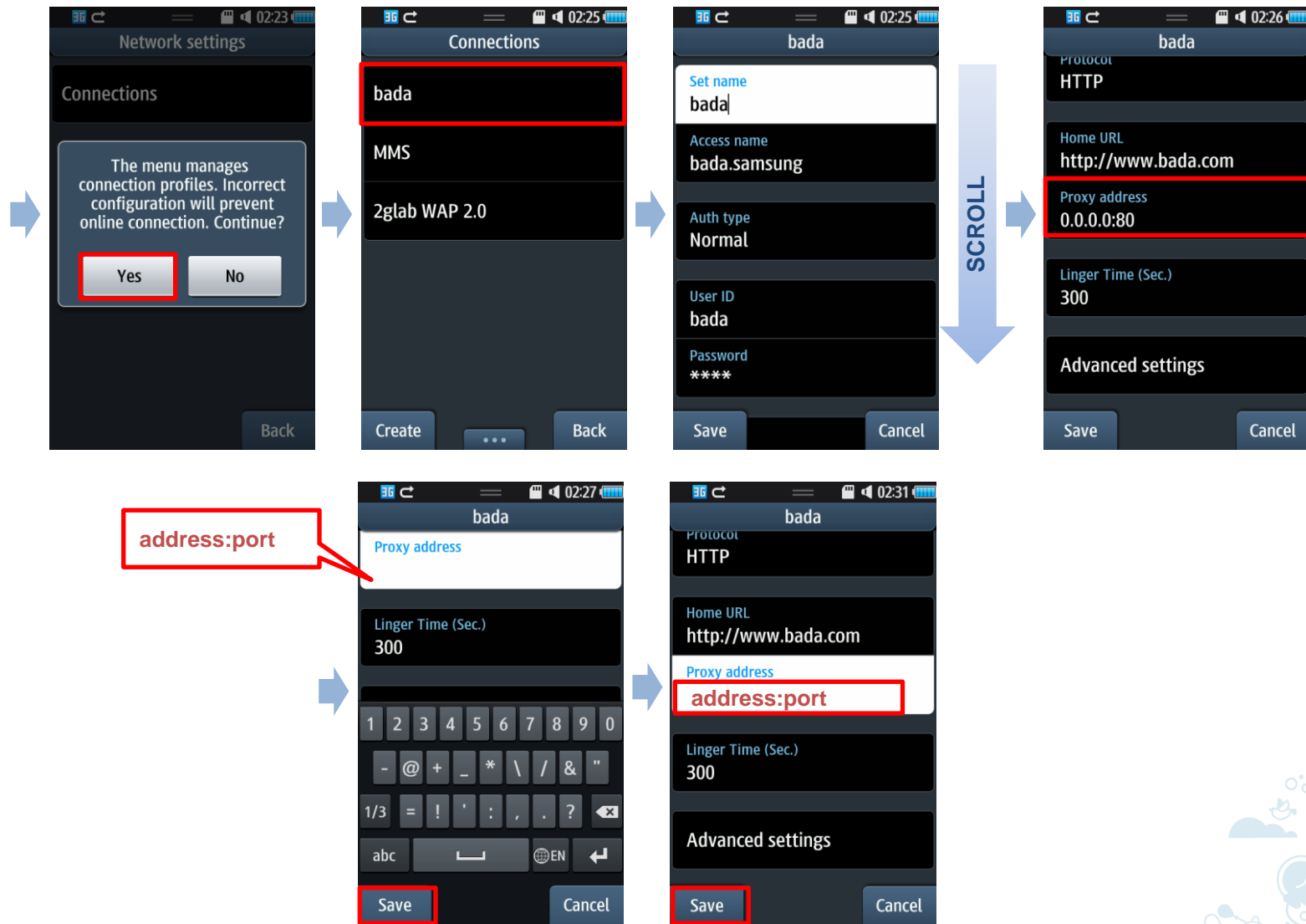


# Proxy Address Setting (1/2)

You can change the proxy address under **Settings > Connectivity > Network > Connections > bada > Proxy address**.



# Proxy Address Setting (2/2)



# Review

1. What is needed before you can make a connection?
2. What class tells you the protocol type?
3. How do you find out an IP address from a fully qualified domain name?
4. Is it possible to get a host name from an IP address in bada?
5. `NetStatistics` provides some traffic information. What is it?

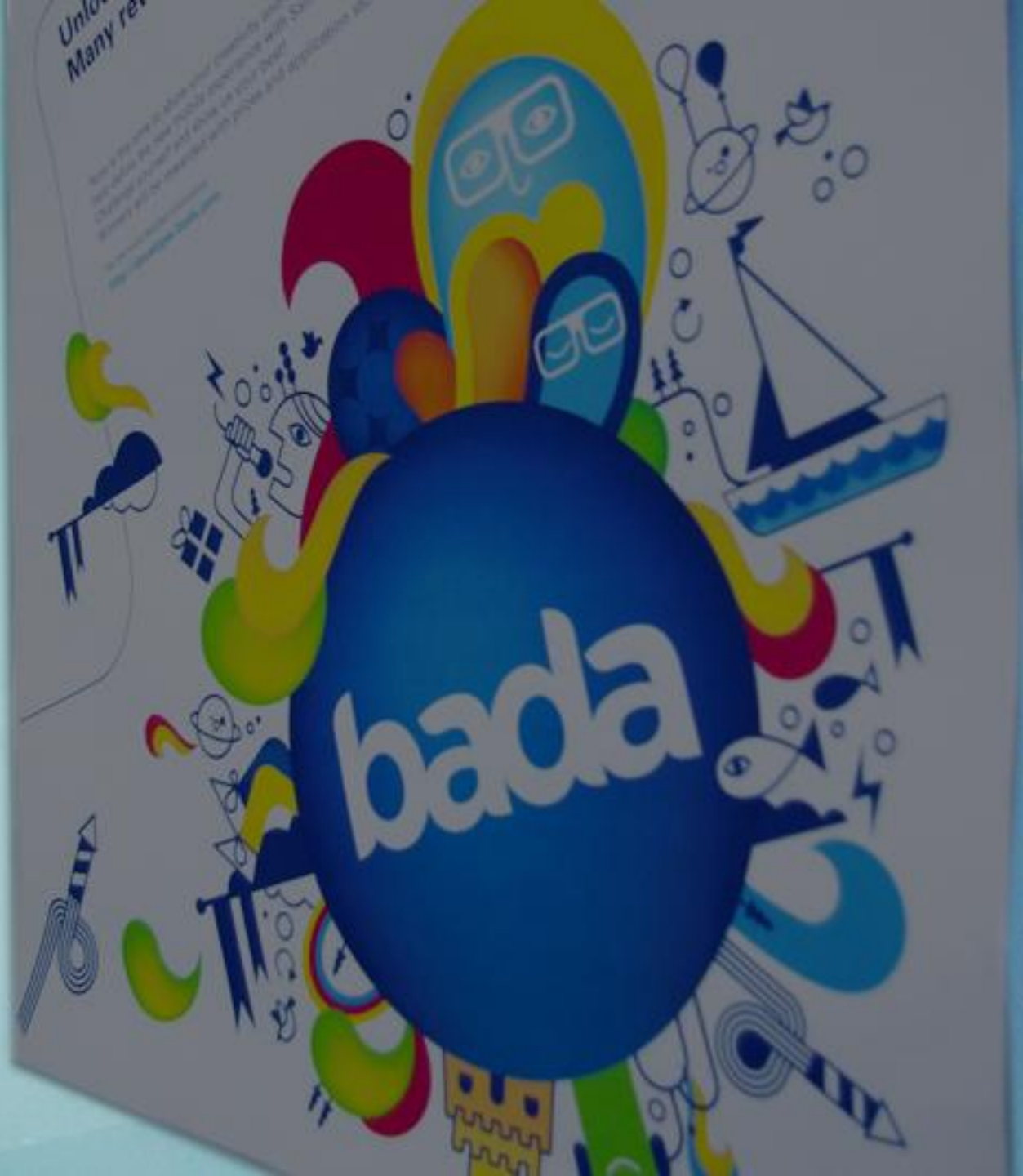


# Answers

1. You need a network account first. One way to get a net account is through 1 of the the `NetAccountManager::GetNetAccount~()` methods.
2. `NetConnectionInfo`.
3. You use the `Dns::GetHostByName()` method.
4. Yes. `Dns::GetHostByAddress()` returns a host name.
5. `NetStatistics` provides the size of the last sent or received data in bytes as well as the total of all traffic in bytes.



# Sockets



# Contents

- Essential Classes
- Relationships between Classes
- Overview
- Non-blocking mode
  - Example: TCP Client in Non-blocking Mode
- Blocking mode
  - Example: UDP Client in Blocking Mode
- Secure Sockets
  - Example: Secure Sockets
- Sample Application: "SocketChat"
- FAQ
- Review
- Answers

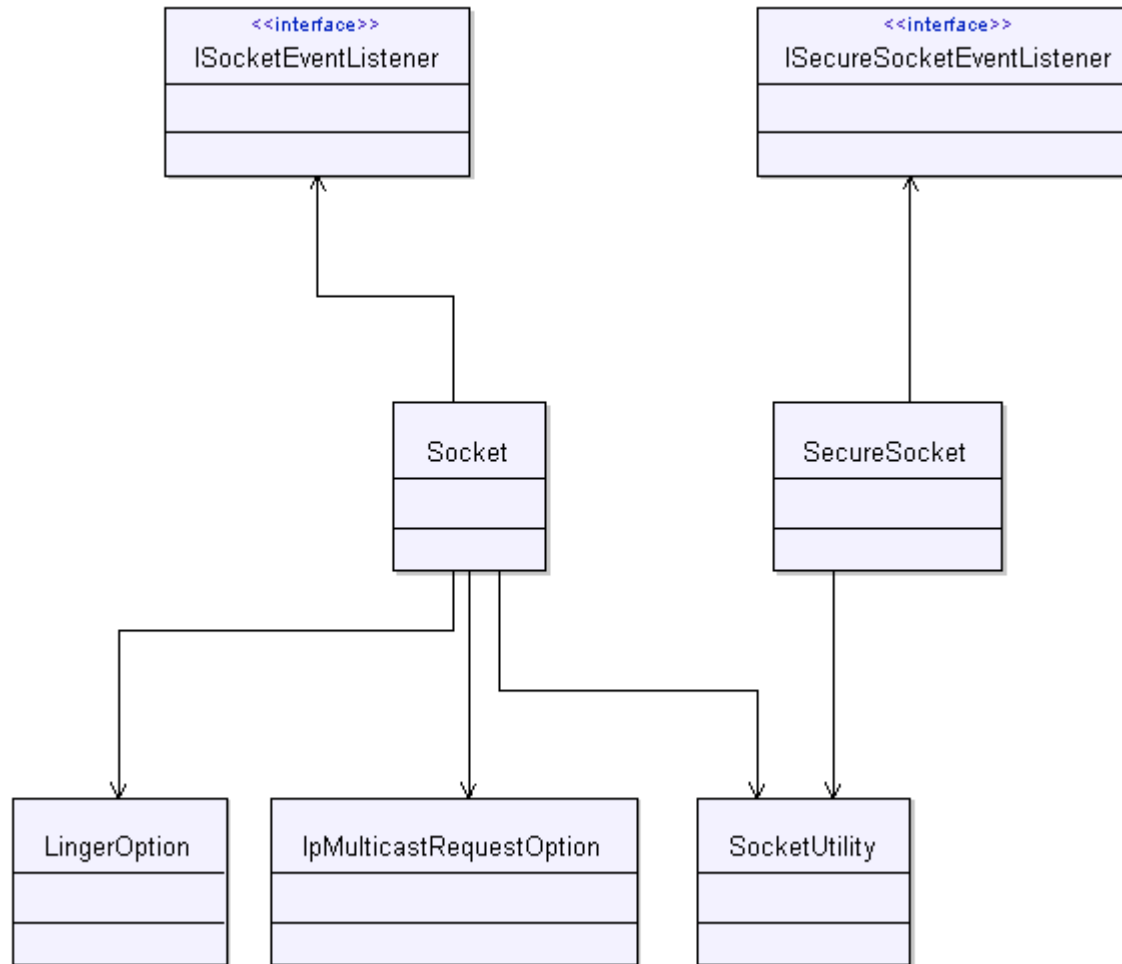


# Essential Classes

Feature	Provided by
Provides BSD-like socket call functionality to communicate with nodes on a network.	Socket
Provides a listener for socket events.	ISocketEventListener
Provides BSD-like secure socket call functionality to communicate with nodes on a network.	SecureSocket
Provides a listener for secure socket events.	ISecureSocketEventListener
Specifies whether a socket remains connected after a <code>Close</code> call and the length of time it remains connected, if data remains to be sent.	LingerOption
Provides support for multicasting in sockets.	IpMulticastRequestOption
Provides socket utilities, such as <code>select</code> , <code>host</code> , and network conversion.	SocketUtility



# Relationships between Classes



# Overview

- A socket is one endpoint in a two-way communication link between two programs running on a network.
- The `Socket` and `SecureSocket` classes provide a rich set of methods and properties for network communications. These classes allow you to communicate over a network.
- Key features include:
  - BSD-like socket handling
  - Sockets:
    - TCP client, TCP server, and UDP client
    - Non-blocking (default) and blocking modes
  - Secure sockets:
    - Support for TLS1.0 and SSL 3.0
    - TCP client socket functionality only
    - Non-blocking mode only
  - Key methods: `Listen()`, `AcceptN()`, `Connect()`, `Send()`, `SendTo()`, `Receive()`, `ReceiveFrom()`, and `Close()`.

# Non-blocking Mode

- Sockets are asynchronous (non-blocking mode) by default.
- The basic flow for using non-blocking sockets is an event-driven procedure.
- Non-blocking mode can be used with both TCP and UDP sockets.
- The `ISocketEventListener` provides event handlers for socket events, and is added to a socket through the socket's `AddSocketListener()` method.
- Once the listener is set, you must specify which events to listen for using the socket's `AsyncSelectByListener()` method. Events are specified using the bitwise OR operator (pipes: "|"). For example, "EventA | EventC | EventN".



# Example: TCP Client in Non-blocking Mode

Create a TCP client that sends and receives data.

- Open

```
\<BADA_SDK_HOME>\Examples\Communication\src\Sockets\  
SocketsTcpClientExample.cpp
```

1. **Construct a Socket:**

```
Socket::Construct(NET_SOCKET_AF_IPV4,  
NET_SOCKET_TYPE_STREAM, NET_SOCKET_PROTOCOL_TCP)
```

2. **Add a socket listener:**

```
Socket::AddSocketListener(listener)
```

3. **Set the listener to listen for socket events:**

```
Socket::AsyncSelectByListener(socketEventTypes)
```

4. **Connect to a TCP server:**

```
Socket::Connect(remoteEndPoint)
```

5. **Send and receive data:**

```
Socket::Send(buffer)
```

```
Socket::Receive(buffer)
```

6. **Close the socket:**

```
Socket::Close()
```



# Blocking Mode

- Sockets can be created and used synchronously in blocking mode.
- To use a socket in blocking mode, you must call the `Socket::Ioctl()` method with the `NET_SOCKET_FIONBIO` option after initializing the socket.
- Blocking mode can be used with both TCP and UDP sockets.
- Use caution when using sockets in blocking mode. Some hardware does not support sockets in blocking mode.



# Example: UDP Client in Blocking Mode

Create a UDP client that sends data to other clients and receives data from other clients.

- Open

```
\<BADA_SDK_HOME>\Examples\Communication\src\Sockets\  
SocketsUdpExample.cpp
```

1. **Construct a Socket:**

```
Socket::Construct(Net_SOCKET_AF_IPV4,  
NET_SOCKET_TYPE_DATAGRAM, NET_SOCKET_PROTOCOL_UDP)
```

2. **Send and receive some data:**

```
Socket::SendTo(buffer, remoteEndPoint)  
Socket::ReceiveFrom(buffer, remoteEndPoint)
```

3. **Close the socket:**

```
Socket::Close()
```



# Secure Sockets

- Secure sockets are only used asynchronously in non-blocking mode, and only with TCP.
- Samsung bada includes secure sockets support for TLS1.0 and partial support for SSL3.0.
- The `ISecureSocketEventListener` provides event handlers for secure socket events, and is added to a secure socket through the secure socket's `AddSecureSocketListener()` methods.
- Once the listener is set, you must specify which events to listen for using the socket's `AsyncSelectByListener()` method. Events are specified using the bitwise OR operator (pipes: "|"). For example, "EventA | EventC | EventN".



# Example: Secure Sockets

Create a secure client that sends and receives data.

- Open `<BADA_SDK_HOME>\Examples\Communication\src\Sockets\SocketsSecureExample.cpp`

1. **Construct a SecureSocket:**

```
SecureSocket::Construct(Net_SOCKET_AF_IPV4,  
NET_SOCKET_TYPE_STREAM, NET_SOCKET_PROTOCOL_SSL)
```

2. **Add a secure socket listener:**

```
SecureSocket::AddSecureSocketListener(listener)
```

3. **Set the listener to listen for secure socket events:**

```
SecureSocket::AsyncSelectByListener(socketEventTypes)
```

4. **Connect to a secure server:**

```
SecureSocket::Connect(remoteEndPoint)
```

5. **Send and receive data:**

```
SecureSocket::Send(buffer)  
SecureSocket::Receive(buffer)
```

6. **Close the secure socket:**

```
SecureSocket::Close()
```



# Sample Application: "SocketChat"

- Open `\<BADA_SDK_HOME>\Samples\SocketChat\src`.
- The SocketChat application shows how to:
  - Create a server socket.
  - Create a client socket.
  - Communicate using UDP or TCP sockets.



# FAQ (1/3)

- What role does the `SocketUtility` class play?
  - It provides network utility tools to perform common tasks, such as:
    - Determining the status of one or more sockets, waiting if necessary.
      - `SocketUtility::Select()`
    - Converting host byte order and network byte order.
      - `SocketUtility::HtoNL()`
      - `SocketUtility::HtoNS()`
      - `SocketUtility::NtoHL()`
      - `SocketUtility::NtoHS()`



# FAQ (2/4)

Which ciphers does the `SecureSocket` class support?

Type	Description
Certificate types	X.509
Protocols	TLS1.0, SSL3.0
Ciphers	AES-256-CBC, AES-128-CBC, 3DES-CBC, DES-ECB, 3DES-ECB, DES-CBC, AES-192-CBC, AES-ECB-128, AES-192-ECB, AES-ECB-256
MACs	SHA512, SHA384, SHA256, SHA1, MD5
Key exchange algorithms	RSA
Compression methods	NULL



# FAQ (3/4)

What options can I use?

Level	Name	Description	Set	Get
IPPROTO_TCP	TCP_NODELAY	Disables the Nagle algorithm for send coalescing.	O	O
	TCP_MAXSEG	The MSS (Maximum Segment Size) for TCP.	O	O
IPPROTO_IP	IP_TTL	The time-to-live.	O	O
	IP_TOS	The type-of-service and precedence.	O	O
	IP_ADD_MEMBERSHIP	Adds membership for multicasting.	O	X
	IP_DROP_MEMBERSHIP	Drops membership for multicasting.	O	X



# FAQ (4/4)

Level	Name	Description	Set	Get
SOL_SOCKET	SO_ACCEPTCONN	The socket is listening.	X	O
	SO_BROADCAST	Permits sending datagrams to broadcast addresses.	O	O
	SO_DEBUG	Records debugging information.	NS	NS
	SO_DONTROUTE	Do not route; send the packet directly to the interface addresses.	NS	NS
	SO_ERROR	Gets and clears any socket errors.	NS	NS
	SO_KEEPALIVE	Sends keep-alives.	O	O
	SO_LINGER	The linger options.	O	O
	SO_OOBINLINE	Enables receiving 'Out Of Band' data inline.	O	O
	SO_RCVBUF	The buffer size for receiving.	O	O
	SO_RCVLOWAT	The low water mark for Receive operations.	NS	NS
	SO_RCVTIMEO	The receive socket buffer time-out value.	O	O
	SO_REUSEADDR	Allows the socket to be bound to an address that is already in use.	O	O
	SO_SNDBUF	The buffer size for sends.	O	O
	SO_SNDLOWAT	The low water mark for Send operations.	NS	NS
	SO_SNDTIMEO	The send socket buffer time-out.	O	O
	SO_TYPE	The socket type.	X	O
	SO_SSLVERSION	The SSL version of the secure socket (only for secure sockets).	O	O
SO_SSLCIPHERSUITEID	The SSL cipher suite ID of the secure socket (only for secure sockets).	O	O	
SO_SSLCERTVERIFY	The SSL certificate verification of the secure socket (only for secure sockets).	O	X	

NS: Not supported.

# Review

1. Fill in the blank: The `SecureSockets` class provides functionality that is better known as \_\_\_\_\_?
2. You need to perform a host to network conversion. What bada class do you use?
3. Which enumeration would you use when working in blocking mode?
  - a) `NetSecureSocketSslCipherSuiteID`
  - b) `NetSecureSocketSslVersion`
  - c) `NetSocketAddressFamily`
  - d) `NetSocketIoctlCmd`
  - e) `NetSocketOptLevel`
  - f) `NetSocketOptName`
  - g) `NetSocketProtocol`
  - h) `NetSocketType`

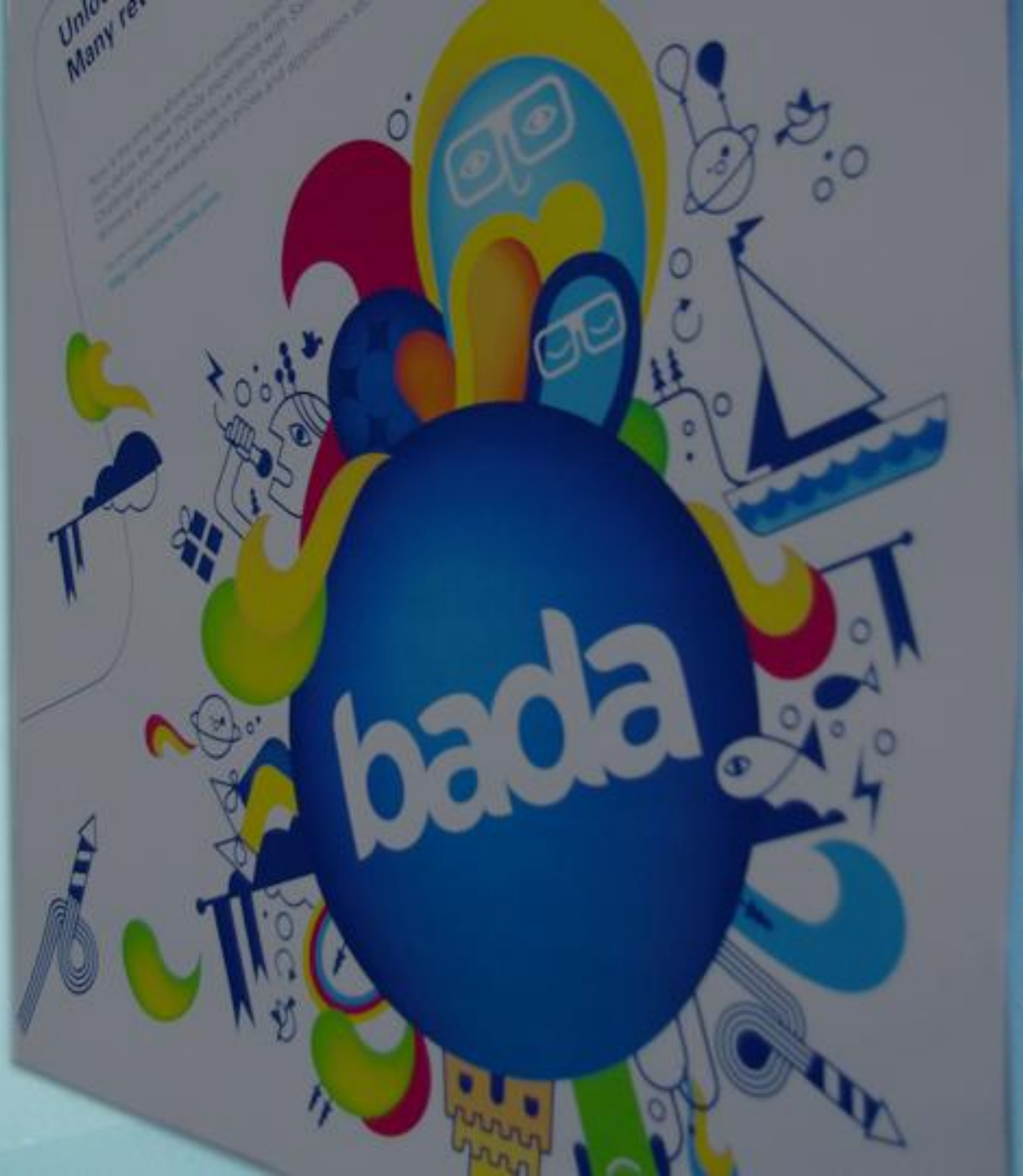


# Answers

1. SSL.
2. SocketUtility.
3. d) NetSocketIoctlCmd.



HTTP



# Contents

- Essential Classes
- Relationships between Classes
- Overview
  - HTTP Session and Transaction
  - HTTP Protocol Basics
  - Transfer Mode
  - HTTP Session Mode
  - Example: Send an HTTP Request
  - Example: Receive a Response
  - Example: Send a Request Using POST
- Sample Application: “HttpClient”
- FAQ
- Review
- Answers

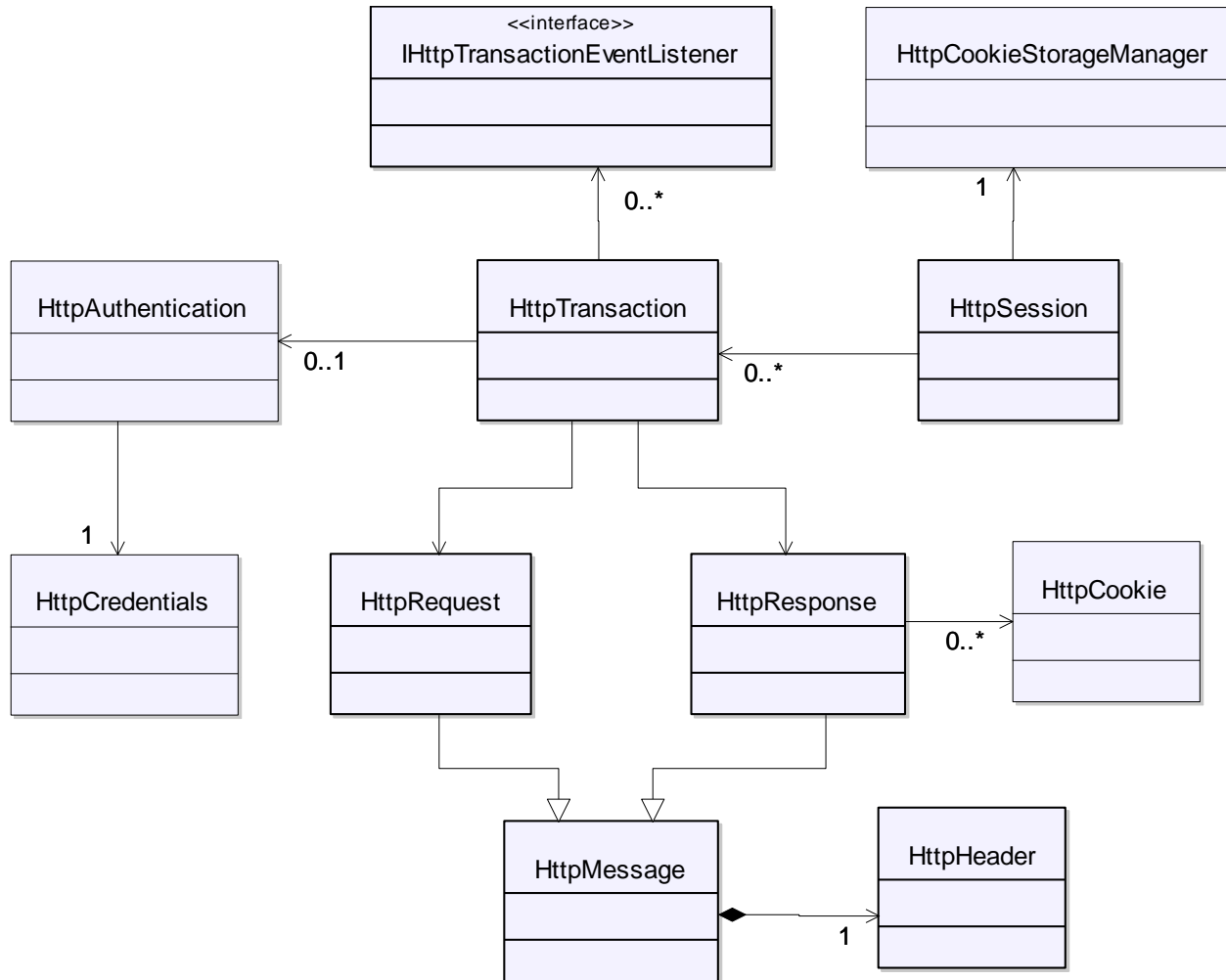


# Essential Classes

Feature	Provided by
Provides a collection of header fields associated with an HTTP message. Provides easy access to headers.	HTTPHeader
Provides the base class for <code>HttpRequest</code> and <code>HttpResponse</code> messages.	HttpMessage
Provides an HTTP request message.	HttpRequest
Provides an HTTP response message.	HttpResponse
Provides an HTTP session object.	HttpSession
Provides an HTTP transaction object that encapsulates <code>HttpRequest</code> and <code>HttpResponse</code> , as well as a listener for <code>HttpTransaction</code> events.	HttpTransaction IHttpTransactionEventListener
Encapsulates cookie functionality for an <code>HttpResponse</code> .	HttpCookie
Manages a collection of HTTP cookies for a session.	HttpCookieStorageManager
Provides tools for HTTP authentication.	HttpAuthentication
Provides tools required for authentication with credentials.	HttpCredentials



# Relationships between Classes



# Overview

- The hypertext transfer protocol is a communications protocol designed to transfer hypertext documents between computers over the World Wide Web. It defines what actions web servers and browsers take in response to various commands.
- Key features include:
  - Multi-Session and Multi-Transaction.
  - Most HTTP 1.1 client features, including pipelining, chunking, and connection management.
  - HTTPS (TLS1.0 and SSL3.0).
  - Full HTTP verb support (HEAD, GET, POST...).
  - HTTP cookies.
  - HTTP authentication:
    - Basic authentication
    - Digest authentication
- “Message” is the HTTP terminology for data exchanged between a client and a server.

# HTTP Session and Transaction

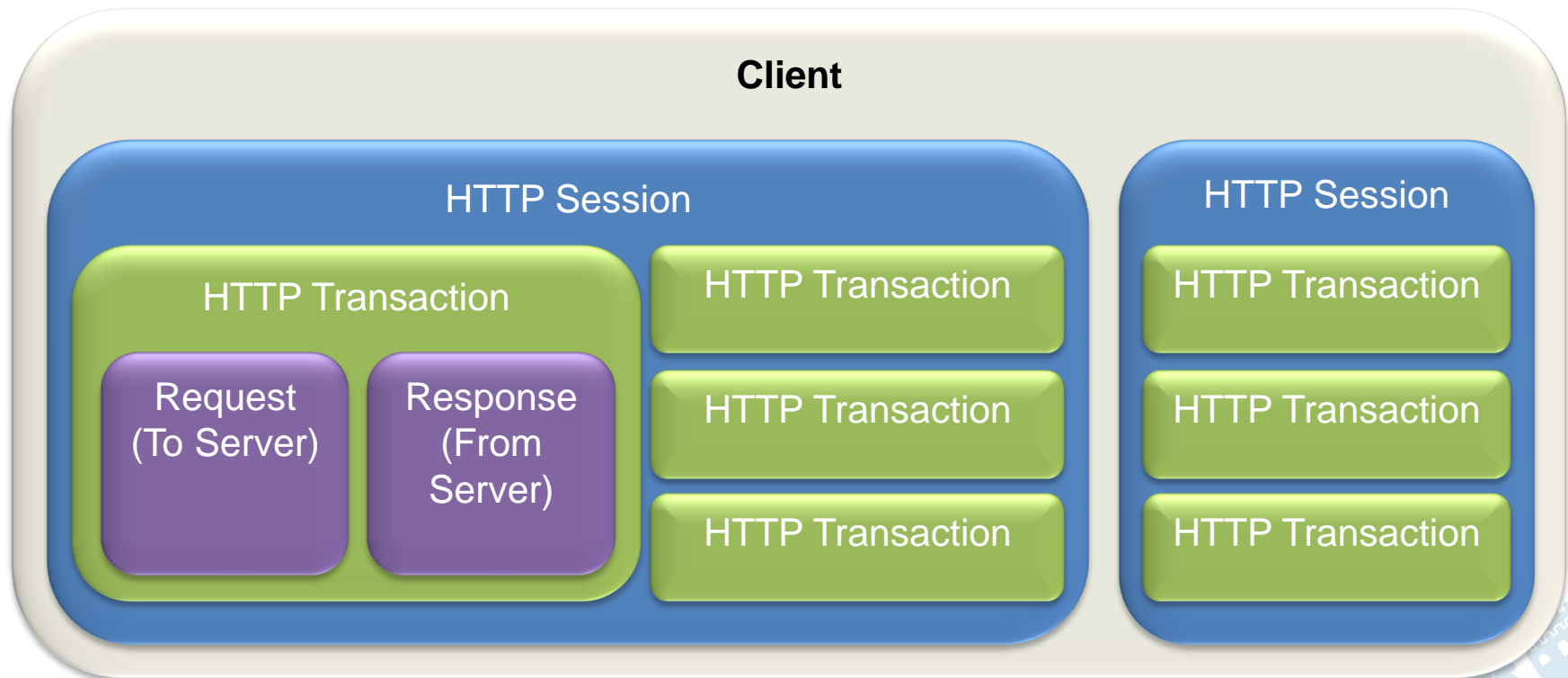
- HTTP Session
  - A session encapsulates the client's HTTP activity over the duration of the client's execution.
  - A set of transactions using the same connection settings, such as a proxy or a host.
- HTTP Transaction
  - A transaction represents an interaction between an HTTP client and an HTTP origin server.
  - A transaction can be submitted in 2 different ways: chunked or non-chunked mode.



# HTTP Protocol Basics

HTTP Sessions > Transactions > Requests and Responses

- Clients have 1+ sessions, and sessions have 1+ transactions.
- Transactions are comprised of a Request and a Response.



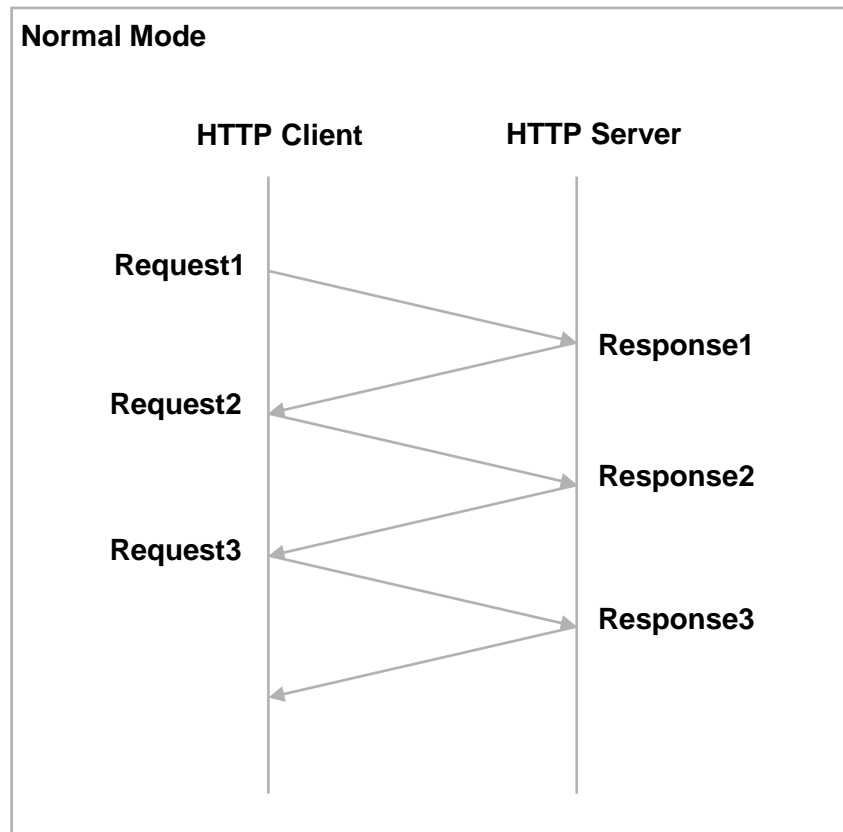
# Transfer Mode

- Data can be sent all at once or in several parts.
- Non-chunked mode:
  - Non-chunked mode is the default for requests.
  - To use the non-chunked mode, add a header field “Content-Length: body-length”.
- Chunked mode:
  - To enable chunking, add a header field “Transfer-encoding: chunked” to a request header and use the `HttpTransaction`’s `EnableTransactionReadyToWrite()` method.
  - Implement an `OnTransactionReadyToWrite()` event handler to send more chunks.
  - An empty chunk is considered to be the last chunk.



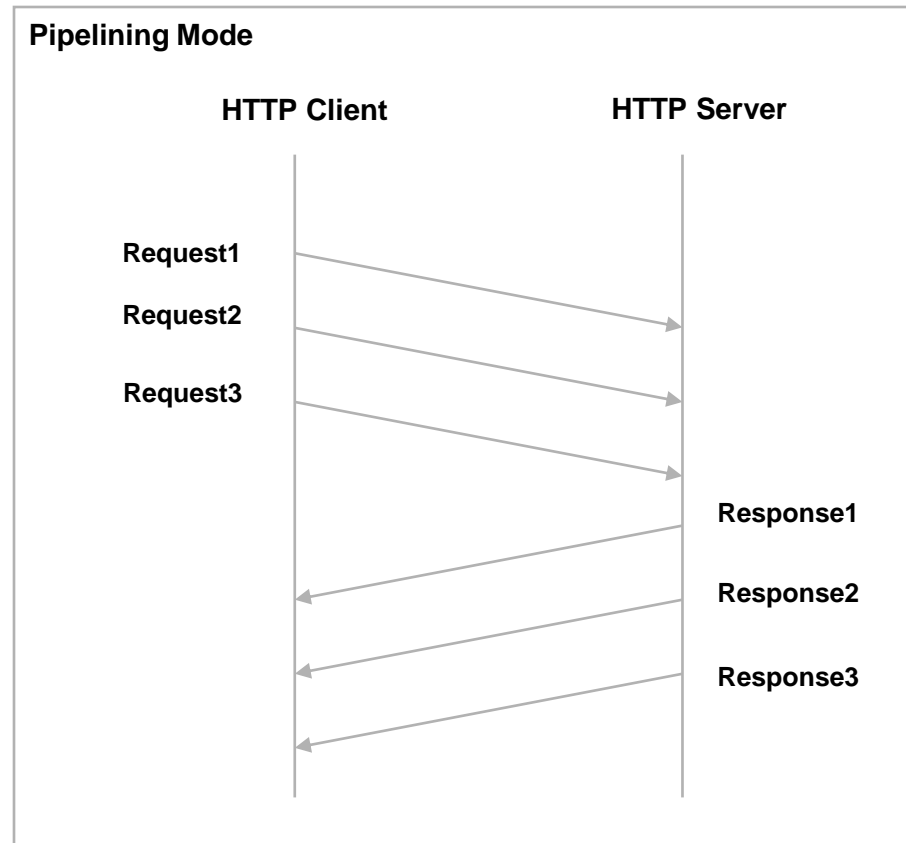
# HTTP Session Mode (1/2)

In normal mode, all HTTP transactions in a session share the same connection. That is, 1 session has 1 connection, or in other words, only one transaction is processed at a time.



# HTTP Session Mode (2/2)

In pipelining mode, as with normal mode, all requests and responses share the same connection, but multiple requests can be submitted concurrently without waiting for each response.



# Example: Send an HTTP Request

## Send an HTTP request.

– Open `<BADA_SDK_HOME>\Examples\Communication\src\Http\HttpExample.cpp, SendRequest()`

1. **Construct an `HttpSession`:**

```
HttpSession::Construct(sessionMode, &proxyAddr,  
hostAddr, commonHeader)
```

2. **Open a new transaction:**

```
HttpSession::OpenTransactionN()
```

3. **Get an HTTP request:**

```
HttpTransaction::GetRequest()
```

4. **Set the HTTP method and URI:**

```
HttpRequest::SetMethod(method)
```

```
HttpRequest::SetUri(uri)
```

5. **Add the HTTP header:**

```
Header::AddField(fieldName, fieldValue)
```

6. **Submit the request:**

```
HttpTransaction::Submit()
```



# Example: Receive a Response

Receive an HTTP response.

- Open `\<BADA_SDK_HOME>\Examples\Communication\src\Http\HttpExample.cpp, OnTransactionReadyToRead()`

1. **Get the response in the event handler:**

```
HttpExample::OnTransactionReadyToRead(session,  
transaction, availableBodyLen)  
HttpTransaction::GetResponse()
```

2. **Read the header:**

```
HttpResponse::GetHeader()
```

3. **Check the status code:**

```
HttpResponse::GetStatusCode()
```

4. **Read the body of the HTTP response:**

```
HttpResponse::ReadBodyN()
```



# Example: Send a Request Using POST (1/2)

## Send a chunked HTTP request using POST.

- Open `<BADA_SDK_HOME>\Examples\Communication\src\http\HttpPostExample.cpp`, `SendPostRequest()` and `OnTransactionReadyToWrite()`

### 1. Get an HTTP request:

```
Net::Http::HttpTransaction::GetRequest()
```

### 2. Add an HTTP header to specify that the request is chunked:

```
Header::AddField(L"Transfer-Encoding", L"chunked")
```

### 3. Enable chunking and the `OnTransactionReadyToWrite()` event handler method:

```
HttpTransaction::EnableTransactionReadyToWrite()
```

### 4. Set the request body:

```
HttpRequest::WriteBody()
```

### 5. Submit the request:

```
HttpTransaction::Submit()
```



# Example: Send a Request Using POST (2/2)

## 6. Send a chunk of the request:

```
HttpPostExample::OnTransactionReadyToWrite()  
WriteBody()
```

## 7. Send the last chunk of the request:

```
HttpPostExample::OnTransactionReadyToWrite()  
WriteBody()
```



# Sample Application: "HttpClient"

- Open `\<BADA_SDK_HOME>\Samples\HttpClient\src`.
  - `HttpClient.cpp`
- HttpClient shows:
  - How to use a session to open a transaction.
  - How to send an HTTP request.
  - How to receive an HTTP response.



# FAQ

- How can I send a request in chunked mode?
  - Add the header field "Transfer-Encoding" with the value "chunked".
  - Call `EnableTransactionReadyToWrite()`.
  - Send chunked data in `OnTransactionReadyToWrite()`.
  - To end the request, send an empty chunk as the last chunk in `OnTransactionReadyToWrite()`.
- How can I set an HTTP proxy?
  - Construct an `HttpSession` instance with the `pProxyAddr` parameter.
- How can I send a request using a persistent connection?
  - Normal and pipelining mode support persistent connections.
- How can I send a request using pipelining?
  - Construct an `HttpSession` instance with `NET_HTTP_SESSION_MODE_PIPELINING`.

# Review

1. Does bada support the OPTIONS and TRACE verbs?
2. What do you call data exchanged between a client and a server in HTTP terminology?
3. True or false. Sessions can have multiple clients.
4. What method do you use to get a request?
5. You have an `HttpRequest*` object, “`pHttpRequest`”, and need to send your request in chunks. How do you turn on chunking?

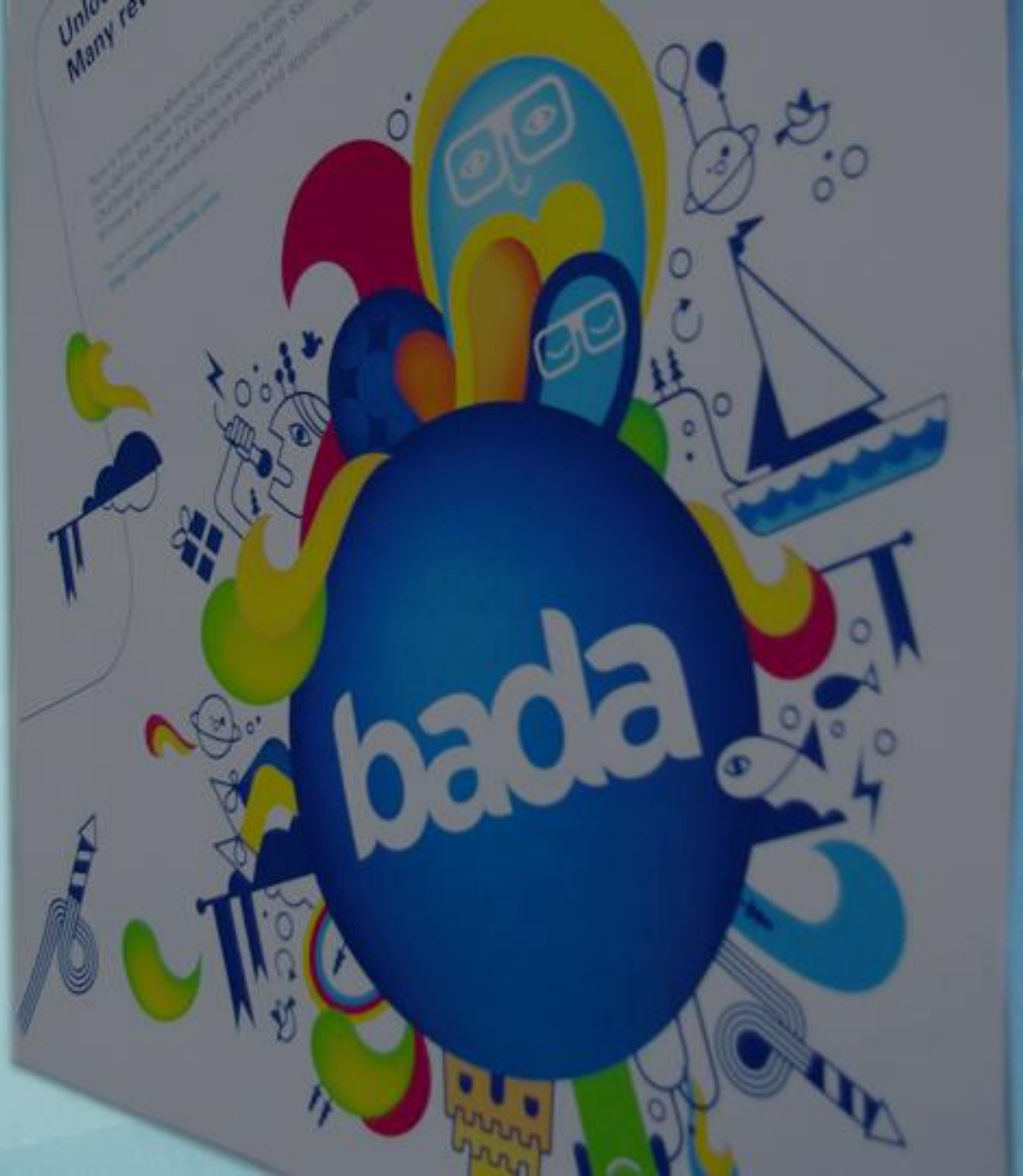


# Answers

1. Yes. The bada Http namespace supports all HTTP verbs.
2. Message.
3. False. The question is nonsense. Clients can have multiple sessions, but sessions cannot have any clients; sessions can have multiple transactions though.
4. `HttpTransaction::GetRequest()`.
5. **Given the `HttpRequest*` object “`pHttpRequest`”, use the `Header::AddField()` method like this:**  

```
HttpHeader* pHeader = null;  
pHeader = pHttpRequest->GetHeader();  
pHeader->AddField(L"Transfer-Encoding",  
L"chunked");
```

Wi-Fi



# Contents

- Essential Classes
- Relationships between Classes
- Overview
- Wi-Fi Modes
- Examples
  - Example: Activate and Deactivate Wi-Fi
  - Example: Get Wi-Fi Account Information
  - Example: Connect with a Specific Access Point after Scanning
  - Example: Start and Stop an Ad Hoc Connection
- Sample Application: "AdhocChatApp"
- FAQ
- Review
- Answers

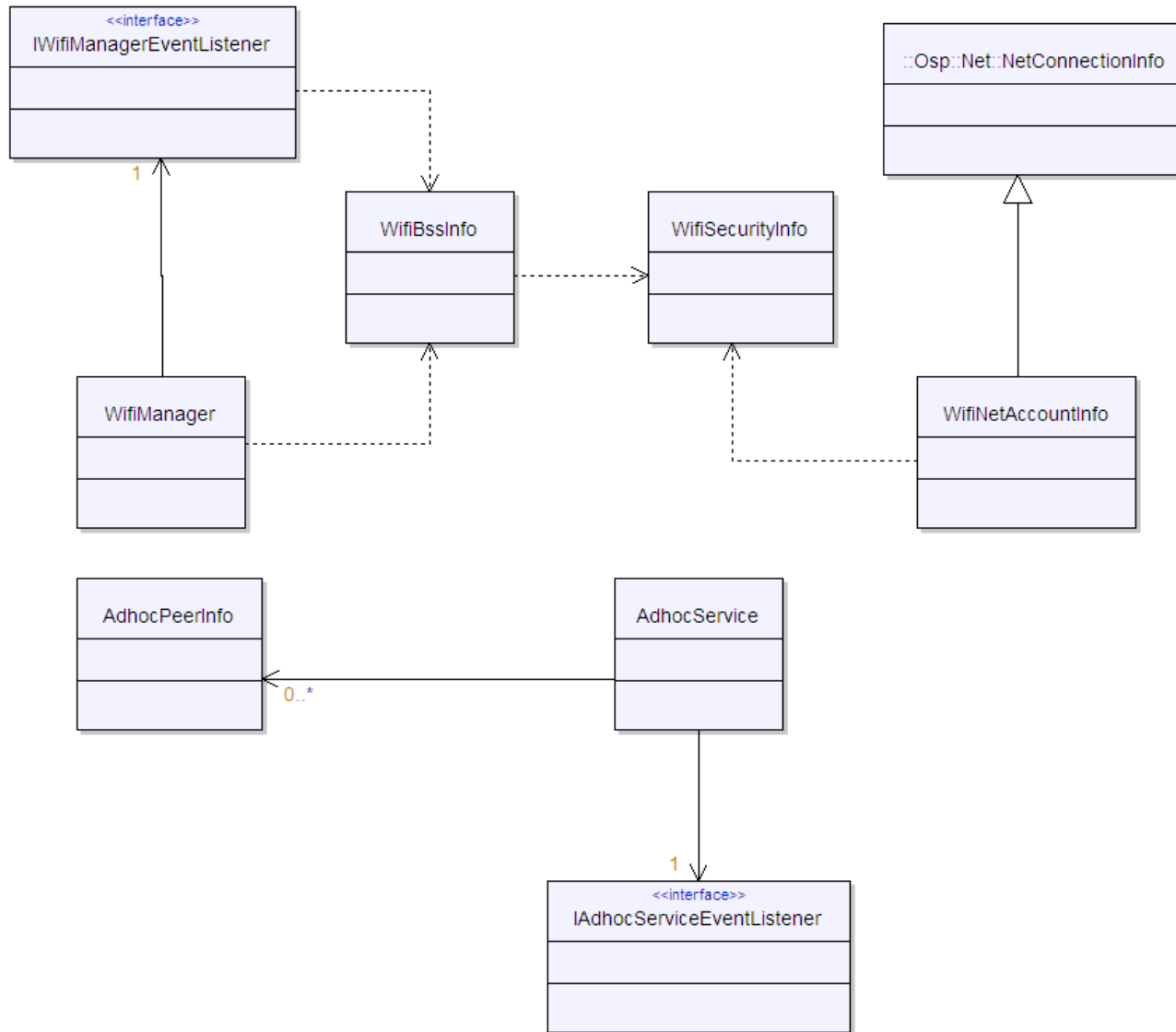


# Essential Classes

Feature	Provided by
Manages the local Wi-Fi device.	WifiManager
Listens for <code>WifiManager</code> events.	<code>IWifiManagerEventListener</code>
Encapsulates all Wi-Fi account information.	<code>WifiNetAccountInfo</code>
Opens Wi-Fi services through an ad hoc Network.	<code>AdhocService</code>
Listens for <code>AdhocService</code> events.	<code>IAdhocServiceEventListener</code>
Provides a base type for ad hoc peer information.	<code>AdhocPeerInfo</code>
Provides functionality to manage Wi-Fi BSS information.	<code>WifiBssInfo</code>
Provides functionality to manage Wi-Fi security information.	<code>WifiSecurityInfo</code>



# Relationships between Classes



# Overview

- The Wi-Fi namespace facilitates activating and deactivating a local Wi-Fi device and establishing an ad hoc network.
- Key features include:
  - Wi-Fi device management
  - Wi-Fi connection management
    - Infrastructure mode
    - Independent mode (ad hoc mode)
  - Wi-Fi account info management



# Wi-Fi Modes

Wi-Fi can connect in two modes:

- Infrastructure mode
  - A wireless access point is required for infrastructure mode wireless networking.
  - To join the wireless local area network (WLAN), a client must be configured to use the same service set identifier (SSID) as the access point.
- Independent mode (ad hoc mode)
  - This mode allows a Wi-Fi network to function without a central wireless router or access point.
  - All wireless peers on the ad hoc network must use the same SSID and channel number. Currently, the channel is not controllable, the Wi-Fi device scans and selects a proper channel by itself.
  - `AdhocService` helps communicate over ad hoc networks:
    - Provides simple methods to create and manage ad hoc networks.
    - Provides peer information in the same ad hoc network.

# Example: Activate and Deactivate Wi-Fi

Activate and deactivate a Wi-Fi connection.

– Open

```
\<BADA_SDK_HOME>\Examples\Communication\src\Wifi\
WifiExample.cpp, ActivateDeactivateWiFi()
```

1. **Create a WifiManager:**

```
WifiManager::Construct(listener)
```

2. **Activate Wi-Fi:**

```
WifiManager::Activate()
```

3. **Check that Wi-Fi is active:**

```
IWifiManagerEventListener::OnWifiActivated(r)
```

4. **Check that Wi-Fi is connected:**

```
IWifiManagerEventListener::OnWifiConnected(ssid, r)
```

5. **Deactivate the Wi-Fi connection:**

```
WifiManager::Deactivate()
```

# Example: Get Wi-Fi Account Information

Get Wi-Fi account information.

– Open

```
\<BADA_SDK_HOME>\Examples\Communication\src\Wifi\  
WifiExample.cpp, GetWifiAccountInfo()
```

1. **Get a Wi-Fi network account ID (the valid Wi-Fi ID):**

```
NetAccountManager::GetNetAccountId(NET_BEARER_WIFI)
```

2. **Get network account information:**

```
NetAccountManager::GetNetAccountInfoN()
```

3. **Check if the obtained NetAccountInfo is about Wi-Fi:**

```
WifiNetAccountInfo::GetBearerType()
```

4. **Get information from the Wi-Fi account info:**

```
WifiNetAccountInfo::
```

- GetBssId()
- GetBssType()
- GetRadioChannel()
- GetSecurityInfo()
- GetSsid()

# Example: Connect with a Specific Access Point after Scanning

Scan nearby access points (AP) and connect with a specific AP.

- Open `<BADA_SDK_HOME>\Examples\Communication\src\Wifi\WifiExample.cpp, ConnectAfterScanning()`

## 1. Scan nearby APs:

- Scan for APs:

```
WifiManager::Scan()
```

- Get `WifiBssInfo` through the `OnWifiScanCompletedN()` event handler.

## 2. Connect with an AP:

- Get the security info:

```
WifiBssInfo::GetSecurityInfo()
```

- Set the network key according to the encryption type of the target AP:

```
WifiSecurityInfo::
```

- `SetNetworkKeyWep()`
- `SetNetworkKeyTkip()`
- `SetNetworkKeyAes()`

- Connect with the AP:

```
WifiManager::Connect(WifiBssInfo&)
```

# Example: Start and Stop an Ad Hoc Connection

Illustrates how to use the `AdhocService`.

– Open

```
\<BADA_SDK_HOME>\Examples\Communication\src\Wifi\
WifiExample.cpp, AdhocApp ()
```

1. **Start the ad hoc service:**

```
AdhocService::StartAdhocService ()
```

2. **Get information about neighbors:**

```
AdhocService::GetNeighborsN ()
```

3. **Broadcast a message to neighbors:**

```
AdhocService::SendBroadcastMessage ()
```

4. **Stop the ad hoc service:**

```
AdhocService::StopAdhocService ()
```



# Sample Application: “AdhocChatApp”

- Open \<BADA\_SDK\_HOME>\Samples\AdhocChatApp\src.
- The AdhocChatApp application shows how to:
  - Use Wi-Fi ad hoc services.
  - Send and receive unicast messages in a Wi-Fi ad hoc network.
  - Send and receive broadcast messages in a Wi-Fi ad hoc network.



# FAQ

- How do I create a Wi-Fi net account?
  - You cannot programmatically create, update, or delete a Wi-Fi net account through the `NetAccountManager` or any bada methods.
  - You can only get the Wi-Fi net account information from `NetAccountManager`.
  - You can have the user add or delete Wi-Fi information through the Wi-Fi Setting UI.
- How many peers can I have in the Wi-Fi ad hoc mode in bada simultaneously?
  - Samsung bada manages up to 8 peers simultaneously.



# Review

1. What are the two modes that Wi-Fi can connect in?
2. Does the `WifiManager` use 1-phase or 2-phase construction?
3. True or false. The only limit to the number of neighbors you can see is determined by the physical hardware.
4. True or false. To see if a Wi-Fi connection is active, you use the `WifiManager::Activate()` method.



# Answers

1. Infrastructure mode and independent mode (ad hoc mode).
2. 2-phase construction.
3. False. The `AdhocService` will only allow you to see up to 8 neighbors at a time.
4. False. `WifiManager::Activate()` activates a Wi-Fi connection. You use the `WifiManager::IsConnected()` method to check directly or the `IWifiManagerEventListener::OnWifiConnected()` method to be notified when the Wi-Fi connection is established.



# Contents

- Essential Classes
- Relationships between Classes
- Overview
- Pairing
- Bluetooth Profiles
- Examples
  - Example: Query Bluetooth Devices
  - Example: OPP Client
  - Example: OPP Server
  - Example: SPP Initiator
  - Example: SPP Acceptor
  - Example: Use Application Controls to Pair Bluetooth Devices
- FAQ
- Review
- Answers

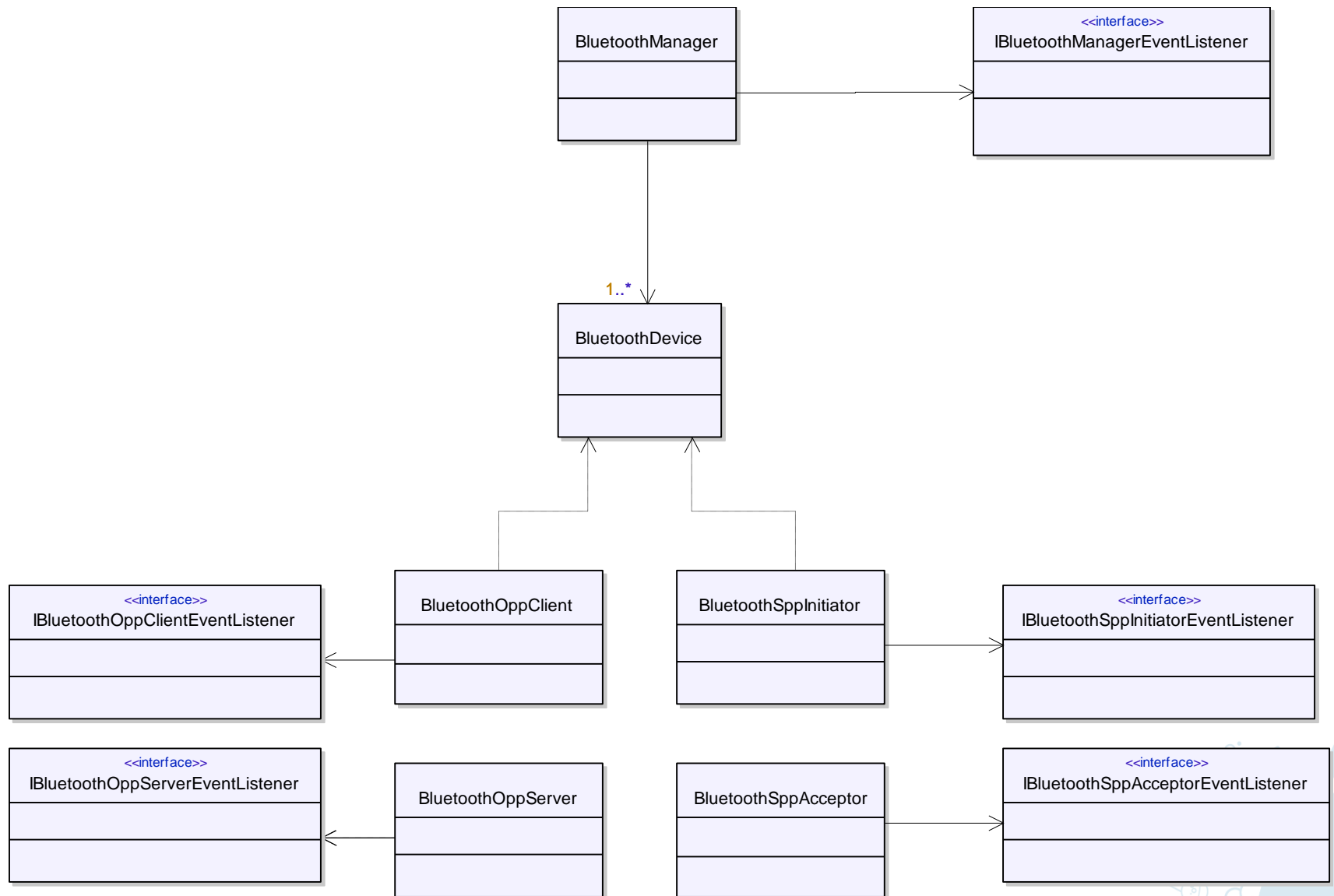


# Essential Classes

Feature	Provided by	
Provides functionality to get information about devices and discovers objects as well as configures the Bluetooth stack.	BluetoothManager BluetoothDevice IBluetoothManagerEventListener	
Handles Bluetooth OPP server, such as incoming push requests.	BluetoothOppServer IBluetoothOppServerEventListener	} OPP
Handles Bluetooth OPP client, such as outgoing push requests.	BluetoothOppClient IBluetoothOppClientEventListener	
Handles Bluetooth SPP acceptor, such as incoming connection requests.	BluetoothSppAcceptor IBluetoothSppAcceptorEventListener	} SPP
Handles Bluetooth SPP initiator, such as outgoing connection requests.	BluetoothSppInitiator IBluetoothSppInitiatorEventListener	



# Relationships between Classes



# Overview

- Bluetooth is a wireless technology that operates in a secure, globally unlicensed Industrial, Scientific and Medical (ISM) 2.4 GHz short-range radio frequency bandwidth.
- Modest performance
  - Up to 723.2 kbps data rate (v1.2).
  - The peak data rate with EDR is 3 Mbps (v2.1).
- Devices can be master and slave simultaneously.
- Currently bada supports only point-to-point connections.



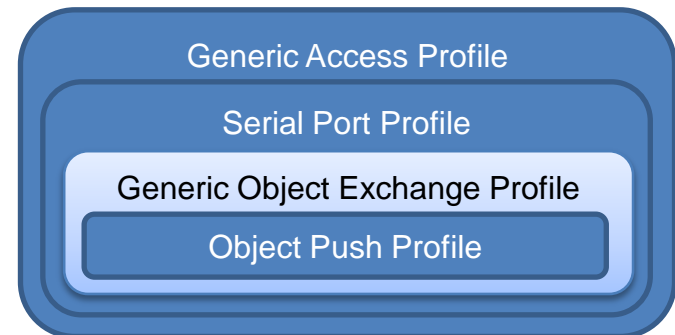
# Pairing

- Pairs of devices can establish a relationship by creating a shared secret known as a link key.
- A device that wants to communicate only with a bonded device can cryptographically authenticate the identity of the other device, and so be sure that it is the same device it previously paired with.
- Link keys can be deleted at any time by either device.
- Pairing process is started internally if you call `BluetoothOppClient::PushFile()` or `BluetoothSppInitiator::Connect()` with the discovered device.



# Bluetooth Profiles (1/2)

- Bluetooth profiles facilitate various connection methods for different devices.
- All Bluetooth profiles derive from the Generic Access Profile (GAP). That is, GAP is the superset for all Bluetooth profiles.
- Bluetooth profile hierarchy of bada supported profiles:



# Bluetooth Profiles (2/2)

supported Bluetooth profiles:

- Generic Access Profile (GAP)
  - Discovers and establishes a connection with other devices (pairing).
  - Provides the basis for all other profiles.
- Serial Port Profile (SPP)
  - SPP is for replacing cables in RS-232 communications with a simple wireless alternative. It is the basis for DUN, FAX, HSP and AVRCP profiles.
- Object Push Profile (OPP)
  - OPP is for sending media and content, such as vCards and appointments. OPP initiates sending content (“push”) rather than having the recipient request it.



# Example: Query Bluetooth Devices

Get information about Bluetooth devices.

- Open `<\BADA_SDK_HOME>\Examples\Communication\src\Bluetooth\BluetoothExample.cpp, GetBluetoothDevice()`

## 1. Activate Bluetooth:

```
BluetoothManager::Activate()
```

## 2. Get information about Bluetooth devices:

```
BluetoothManager::GetLocalDevice()
```

```
BluetoothManager::GetPairedDeviceList()
```

```
BluetoothManager::GetPairedDeviceAt(index)
```

## 3. Deactivate Bluetooth:

```
BluetoothManager::Deactivate()
```



# Example: OPP Client

Use a Bluetooth device as an OPP client.

- Open `<BADA_SDK_HOME>\Examples\Communication\src\Bluetooth\BluetoothExample.cpp`,  
`BluetoothOppClientExample()`

## 1. Activate Bluetooth:

```
Bluetooth::BluetoothManager::Activate()
```

## 2. Get a Bluetooth device:

```
BluetoothManager::GetLocalDevice()
```

```
BluetoothManager::GetPairedDeviceList()
```

```
BluetoothManager::GetPairedDeviceAt(index)
```

## 3. Construct an OPP client:

```
BluetoothOppClient::Construct(listener)
```

## 4. Push the file to other Bluetooth devices (OPP server):

```
BluetoothOppClient::PushFile(remoteDevice, filePath)
```

# Example: OPP Server

Use a Bluetooth device as an OPP server.

- Open `<BADA_SDK_HOME>\Examples\Communication\src\Bluetooth\BluetoothExample.cpp`,  
`BluetoothOppServerExample()`

1. **Activate Bluetooth:**

```
BluetoothManager::Activate()
```

2. **Construct an OPP server:**

```
BluetoothOppServer::Construct(listener)
```

3. **Start the OPP service:**

```
BluetoothOppServer::StartService(dstPath)
```

4. **Accept a connection:**

```
BluetoothOppServer::AcceptPush()
```

5. **Stop the OPP service:**

```
BluetoothOppServer::StopService()
```



# Example: SPP Initiator

Use a Bluetooth device as an SPP initiator.

- Open `<BADA_SDK_HOME>\Examples\Communication\src\Bluetooth\BluetoothExample.cpp`,  
`BluetoothSppInitiatorExample()`

## 1. Activate Bluetooth:

```
BluetoothManager::Activate()
```

## 2. Get a Bluetooth device:

```
BluetoothManager::GetLocalDevice()
```

```
BluetoothManager::GetPairedDeviceList()
```

```
BluetoothManager::GetPairedDeviceAt(index)
```

## 3. Construct an SPP initiator:

```
BluetoothSppInitiator::Construct(listener)
```

## 4. Connect to an SPP acceptor:

```
BluetoothSppInitiator::Connect(remoteDevice)
```

## 5. Send some data:

```
BluetoothSppInitiator::SendData(byteBuffer)
```

# Example: SPP Acceptor

Use a Bluetooth device as an SPP Acceptor.

- Open `<BADA_SDK_HOME>\Examples\Communication\src\Bluetooth\BluetoothExample.cpp`,  
`BluetoothSppAcceptorExample()`

**1. Activate Bluetooth:**

```
BluetoothManager::Activate()
```

**2. Construct an SPP acceptor:**

```
BluetoothSppAcceptor::Construct(listener)
```

**3. Start the SPP service:**

```
BluetoothSppAcceptor::StartService()
```

**4. Accept a connection:**

```
BluetoothSppAcceptor::AcceptConnection()
```

**5. Stop the SPP service:**

```
BluetoothSppAcceptor::StopService()
```



# Example: Use Application Controls to Pair Bluetooth Devices

Get a specific device through application controls and start the pairing process by pushing a file to the device.

- Open `\<BADA_SDK_HOME>\Examples\Communication\src\Bluetooth\BluetoothExample.cpp`, `DiscoverPairAndPush()`

## 1. Get a specific device through application controls:

- Get an `AppControl` instance and start the application control behavior:

```
AppManager::FindAppControlN(APPCONTROL_BT,  
OPERATION_PICK);  
AppControl::Start();
```

- Create an instance of `BluetoothDevice` using the `pResultList` parameter of the `OnAppControlCompleted()` event handler:

```
remoteDevice =  
BluetoothDevice::GetInstanceFromAppControlResultN  
(*pResultList);
```

## 2. Push a file to the created `BluetoothDevice` to start pairing:

```
BluetoothOppClient::PushFile(remoteDevice, filePath)
```

The pairing process is started internally before the push request is sent.

# FAQ

- How can I pair Bluetooth devices?
  - There is no direct API for pairing devices. However, you can use application controls to discover a new device. The pairing process is started internally if you call `BluetoothOppClient::PushFile()` or `BluetoothSppInitiator::Connect()` with the discovered device.
- Can I use all Bluetooth profiles?
  - Samsung bada only supports the GAP, SPP, and OPP profiles.



# Review

1. True or false. bada supports all Bluetooth profiles.



# Answers

1. False. bada only supports the GAP, SPP and OPP profiles at present.





**bada**

The platform with more opportunities  
Invitation to Adventure