# Chaincode Tutorials

## What is Chaincode?

Chaincode is a program, written in [Go](https://golang.org/), [node.js](https://nodejs.org/), or [Java](https://java.com/en/) that implements a prescribed interface. Chaincode runs in a secured Docker container isolated from the endorsing peer process. Chaincode initializes and manages ledger state through transactions submitted by applications.

A chaincode typically handles business logic agreed to by members of the network, so it may be considered as a “smart contract”. State created by a chaincode is scoped exclusively to that chaincode and can’t be accessed directly by another chaincode. However, within the same network, given the appropriate permission a chaincode may invoke another chaincode to access its state.

## Two Personas

We offer two different perspectives on chaincode. One, from the perspective of an application developer developing a blockchain application/solution entitled [Chaincode for Developers](https://hyperledger-fabric.readthedocs.io/en/latest/chaincode4ade.html), and the other, [Chaincode for Operators](https://hyperledger-fabric.readthedocs.io/en/latest/chaincode4noah.html) oriented to the blockchain network operator who is responsible for managing a blockchain network, and who would leverage the Hyperledger Fabric API to install and govern chaincode, but would likely not be involved in the development of a chaincode application.

## Fabric Chaincode Lifecycle

The Fabric Chaincode Lifecycle is responsible for managing the installation of chaincodes and the definition of their parameters before a chaincode is used on a channel. Starting with Fabric 2.0, governance for chaincodes is fully decentralized: multiple organizations can use the Fabric Chaincode Lifecycle to come to agreement on the parameters of a chaincode, such as the chaincode endorsement policy, before the chaincode is used to interact with the ledger.

The new model offers several improvements over the previous lifecycle:

* **Multiple organizations must agree to the parameters of a chaincode:** In the release 1.x versions of Fabric, one organization had the ability to set parameters of a chaincode (for instance the endorsement policy) for all other channel members. The new Fabric chaincode lifecycle is more flexible since it supports both centralized trust models (such as that of the previous lifecycle model) as well as decentralized models requiring a sufficient number of organizations to agree on an endorsement policy before it goes into effect.
* **Safer chaincode upgrade process:** In the previous chaincode lifecycle, the upgrade transaction could be issued by a single organization, creating a risk for a channel member that had not yet installed the new chaincode. The new model allows for a chaincode to be upgraded only after a sufficient number of organizations have approved the upgrade.
* **Easier endorsement policy updates:** Fabric lifecycle allows you to change an endorsement policy without having to repackage or reinstall the chaincode. Users can also take advantage of a new default policy that requires endorsement from a majority of members on the channel. This policy is updated automatically when organizations are added or removed from the channel.
* **Inspectable chaincode packages:** The Fabric lifecycle packages chaincode in easily readable tar files. This makes it easier to inspect the chaincode package and coordinate installation across multiple organizations.
* **Start multiple chaincodes on a channel using one package:** The previous lifecycle defined each chaincode on the channel using a name and version that was specified when the chaincode package was installed. You can now use a single chaincode package and deploy it multiple times with different names on the same or different channel.

To learn how more about the new Fabric Lifecycle, visit [Chaincode for Operators](https://hyperledger-fabric.readthedocs.io/en/latest/chaincode4noah.html).

You can use the Fabric chaincode lifecycle by creating a new channel and setting the channel capabilities to V2\_0. You will not be able to use the old lifecycle to install, instantiate, or update a chaincode on channels with V2\_0 capabilities enabled. However, you can still invoke chaincode installed using the previous lifecycle model after you enable V2\_0 capabilities.

# Chaincode for Developers

## What is Chaincode?

Chaincode is a program, written in [Go](https://golang.org/), [node.js](https://nodejs.org/), or [Java](https://java.com/en/) that implements a prescribed interface. Chaincode runs in a secured Docker container isolated from the endorsing peer process. Chaincode initializes and manages the ledger state through transactions submitted by applications.

A chaincode typically handles business logic agreed to by members of the network, so it similar to a “smart contract”. A chaincode can be invoked to update or query the ledger in a proposal transaction. Given the appropriate permission, a chaincode may invoke another chaincode, either in the same channel or in different channels, to access its state. Note that, if the called chaincode is on a different channel from the calling chaincode, only read query is allowed. That is, the called chaincode on a different channel is only a Query, which does not participate in state validation checks in subsequent commit phase.

In the following sections, we will explore chaincode through the eyes of an application developer. We’ll present a simple chaincode sample application and walk through the purpose of each method in the Chaincode Shim API.

## Chaincode API

Every chaincode program must implement the Chaincode interface whose methods are called in response to received transactions. You can find the reference documentation of the Chaincode Shim API for different languages below:

* [Go](https://godoc.org/github.com/hyperledger/fabric-chaincode-go/shim#Chaincode)
* [node.js](https://fabric-shim.github.io/ChaincodeInterface.html)
* [Java](https://hyperledger.github.io/fabric-chaincode-java/master/api/org/hyperledger/fabric/shim/Chaincode.html)

In each language, the Invoke method is called by clients to submit transaction proposals. This method allows you to use the chaincode to read and write data on the channel ledger.

You also need to include an Init method that will serve as the initialization function for your chaincode. This method will be called in order to initialize the chaincode when it is started or upgraded. By default, this function is never executed. However, you can use the chaincode definition to request that the Init function be executed. If execution of Init is requested, fabric will ensure that Init is invoked before any other function and is only invoked once. This option provides you additional control over which users can initialize the chaincode and the ability to add initial data to the ledger. If you are using the peer CLI to approve the chaincode definition, use the --init-required flag to request the execution of the Init function. Then call the Init function by using the peer chaincode invoke command and passing the --isInit flag. If you are using the Fabric SDK for Node.js, visit [How to install and start your chaincode](https://hyperledger.github.io/fabric-sdk-node/master/tutorial-chaincode-lifecycle.html). For more information, see [Chaincode for Operators](https://hyperledger-fabric.readthedocs.io/en/latest/chaincode4noah.html).

The other interface in the chaincode “shim” APIs is the ChaincodeStubInterface:

* [Go](https://godoc.org/github.com/hyperledger/fabric-chaincode-go/shim#ChaincodeStubInterface)
* [node.js](https://fabric-shim.github.io/ChaincodeStub.html)
* [Java](https://hyperledger.github.io/fabric-chaincode-java/master/api/org/hyperledger/fabric/shim/ChaincodeStub.html)

which is used to access and modify the ledger, and to make invocations between chaincodes.

In this tutorial using Go chaincode, we will demonstrate the use of these APIs by implementing a simple chaincode application that manages simple “assets”.

## Simple Asset Chaincode

Our application is a basic sample chaincode to create assets (key-value pairs) on the ledger.

### Choosing a Location for the Code

If you haven’t been doing programming in Go, you may want to make sure that you have [Go Programming Language](https://hyperledger-fabric.readthedocs.io/en/latest/prereqs.html#golang) installed and your system properly configured.

Now, you will want to create a directory for your chaincode application as a child directory of $GOPATH/src/.

To keep things simple, let’s use the following command:

mkdir -p $GOPATH/src/sacc && cd $GOPATH/src/sacc

Now, let’s create the source file that we’ll fill in with code:

touch sacc**.**go

### Housekeeping

First, let’s start with some housekeeping. As with every chaincode, it implements the [Chaincode interface](https://godoc.org/github.com/hyperledger/fabric-chaincode-go/shim#Chaincode) in particular, Init and Invoke functions. So, let’s add the Go import statements for the necessary dependencies for our chaincode. We’ll import the chaincode shim package and the [peer protobuf package](https://godoc.org/github.com/hyperledger/fabric-protos-go/peer). Next, let’s add a struct SimpleAsset as a receiver for Chaincode shim functions.

package main

**import** (

"fmt"

"github.com/hyperledger/fabric-chaincode-go/shim"

"github.com/hyperledger/fabric-protos-go/peer"

)

**//** SimpleAsset implements a simple chaincode to manage an asset

type SimpleAsset struct {

}

### Initializing the Chaincode

Next, we’ll implement the Init function.

**//** Init **is** called during chaincode instantiation to initialize any data**.**

func (t **\***SimpleAsset) Init(stub shim**.**ChaincodeStubInterface) peer**.**Response {

}

**Note**

Note that chaincode upgrade also calls this function. When writing a chaincode that will upgrade an existing one, make sure to modify the Init function appropriately. In particular, provide an empty “Init” method if there’s no “migration” or nothing to be initialized as part of the upgrade.

Next, we’ll retrieve the arguments to the Init call using the [ChaincodeStubInterface.GetStringArgs](https://godoc.org/github.com/hyperledger/fabric-chaincode-go/shim" \l "ChaincodeStub.GetStringArgs) function and check for validity. In our case, we are expecting a key-value pair.

**//** Init **is** called during chaincode instantiation to initialize any

**//** data**.** Note that chaincode upgrade also calls this function to reset

**//** **or** to migrate data, so be careful to avoid a scenario where you

**//** inadvertently clobber your ledger's data!

func (t **\***SimpleAsset) Init(stub shim**.**ChaincodeStubInterface) peer**.**Response {

**//** Get the args **from** the transaction proposal

args :**=** stub**.**GetStringArgs()

**if** len(args) **!=** 2 {

**return** shim**.**Error("Incorrect arguments. Expecting a key and a value")

}

}

Next, now that we have established that the call is valid, we’ll store the initial state in the ledger. To do this, we will call [ChaincodeStubInterface.PutState](https://godoc.org/github.com/hyperledger/fabric-chaincode-go/shim" \l "ChaincodeStub.PutState) with the key and value passed in as the arguments. Assuming all went well, return a peer.Response object that indicates the initialization was a success.

**//** Init **is** called during chaincode instantiation to initialize any

**//** data**.** Note that chaincode upgrade also calls this function to reset

**//** **or** to migrate data, so be careful to avoid a scenario where you

**//** inadvertently clobber your ledger's data!

func (t **\***SimpleAsset) Init(stub shim**.**ChaincodeStubInterface) peer**.**Response {

**//** Get the args **from** the transaction proposal

args :**=** stub**.**GetStringArgs()

**if** len(args) **!=** 2 {

**return** shim**.**Error("Incorrect arguments. Expecting a key and a value")

}

**//** Set up any variables **or** assets here by calling stub**.**PutState()

**//** We store the key **and** the value on the ledger

err :**=** stub**.**PutState(args[0], []byte(args[1]))

**if** err **!=** nil {

**return** shim**.**Error(fmt**.**Sprintf("Failed to create asset: %s", args[0]))

}

**return** shim**.**Success(nil)

}

### Invoking the Chaincode

First, let’s add the Invoke function’s signature.

**//** Invoke **is** called per transaction on the chaincode**.** Each transaction **is**

**//** either a 'get' **or** a 'set' on the asset created by Init function**.** The 'set'

**//** method may create a new asset by specifying a new key**-**value pair**.**

func (t **\***SimpleAsset) Invoke(stub shim**.**ChaincodeStubInterface) peer**.**Response {

}

As with the Init function above, we need to extract the arguments from the ChaincodeStubInterface. The Invoke function’s arguments will be the name of the chaincode application function to invoke. In our case, our application will simply have two functions: set and get, that allow the value of an asset to be set or its current state to be retrieved. We first call [ChaincodeStubInterface.GetFunctionAndParameters](https://godoc.org/github.com/hyperledger/fabric-chaincode-go/shim" \l "ChaincodeStub.GetFunctionAndParameters) to extract the function name and the parameters to that chaincode application function.

**//** Invoke **is** called per transaction on the chaincode**.** Each transaction **is**

**//** either a 'get' **or** a 'set' on the asset created by Init function**.** The Set

**//** method may create a new asset by specifying a new key**-**value pair**.**

func (t **\***SimpleAsset) Invoke(stub shim**.**ChaincodeStubInterface) peer**.**Response {

**//** Extract the function **and** args **from** the transaction proposal

fn, args :**=** stub**.**GetFunctionAndParameters()

}

Next, we’ll validate the function name as being either set or get, and invoke those chaincode application functions, returning an appropriate response via the shim.Success or shim.Error functions that will serialize the response into a gRPC protobuf message.

**//** Invoke **is** called per transaction on the chaincode**.** Each transaction **is**

**//** either a 'get' **or** a 'set' on the asset created by Init function**.** The Set

**//** method may create a new asset by specifying a new key**-**value pair**.**

func (t **\***SimpleAsset) Invoke(stub shim**.**ChaincodeStubInterface) peer**.**Response {

**//** Extract the function **and** args **from** the transaction proposal

fn, args :**=** stub**.**GetFunctionAndParameters()

var result string

var err error

**if** fn **==** "set" {

result, err **=** set(stub, args)

} **else** {

result, err **=** get(stub, args)

}

**if** err **!=** nil {

**return** shim**.**Error(err**.**Error())

}

**//** Return the result **as** success payload

**return** shim**.**Success([]byte(result))

}

### Implementing the Chaincode Application

As noted, our chaincode application implements two functions that can be invoked via the Invoke function. Let’s implement those functions now. Note that as we mentioned above, to access the ledger’s state, we will leverage the [ChaincodeStubInterface.PutState](https://godoc.org/github.com/hyperledger/fabric-chaincode-go/shim#ChaincodeStub.PutState) and [ChaincodeStubInterface.GetState](https://godoc.org/github.com/hyperledger/fabric-chaincode-go/shim#ChaincodeStub.GetState) functions of the chaincode shim API.

**//** Set stores the asset (both key **and** value) on the ledger**.** If the key exists,

**//** it will override the value **with** the new one

func set(stub shim**.**ChaincodeStubInterface, args []string) (string, error) {

**if** len(args) **!=** 2 {

**return** "", fmt**.**Errorf("Incorrect arguments. Expecting a key and a value")

}

err :**=** stub**.**PutState(args[0], []byte(args[1]))

**if** err **!=** nil {

**return** "", fmt**.**Errorf("Failed to set asset: %s", args[0])

}

**return** args[1], nil

}

**//** Get returns the value of the specified asset key

func get(stub shim**.**ChaincodeStubInterface, args []string) (string, error) {

**if** len(args) **!=** 1 {

**return** "", fmt**.**Errorf("Incorrect arguments. Expecting a key")

}

value, err :**=** stub**.**GetState(args[0])

**if** err **!=** nil {

**return** "", fmt**.**Errorf("Failed to get asset: %s with error: %s", args[0], err)

}

**if** value **==** nil {

**return** "", fmt**.**Errorf("Asset not found: %s", args[0])

}

**return** string(value), nil

}

### Pulling it All Together

Finally, we need to add the main function, which will call the [shim.Start](https://godoc.org/github.com/hyperledger/fabric-chaincode-go/shim" \l "Start) function. Here’s the whole chaincode program source.

package main

**import** (

"fmt"

"github.com/hyperledger/fabric-chaincode-go/shim"

"github.com/hyperledger/fabric-protos-go/peer"

)

**//** SimpleAsset implements a simple chaincode to manage an asset

type SimpleAsset struct {

}

**//** Init **is** called during chaincode instantiation to initialize any

**//** data**.** Note that chaincode upgrade also calls this function to reset

**//** **or** to migrate data**.**

func (t **\***SimpleAsset) Init(stub shim**.**ChaincodeStubInterface) peer**.**Response {

**//** Get the args **from** the transaction proposal

args :**=** stub**.**GetStringArgs()

**if** len(args) **!=** 2 {

**return** shim**.**Error("Incorrect arguments. Expecting a key and a value")

}

**//** Set up any variables **or** assets here by calling stub**.**PutState()

**//** We store the key **and** the value on the ledger

err :**=** stub**.**PutState(args[0], []byte(args[1]))

**if** err **!=** nil {

**return** shim**.**Error(fmt**.**Sprintf("Failed to create asset: %s", args[0]))

}

**return** shim**.**Success(nil)

}

**//** Invoke **is** called per transaction on the chaincode**.** Each transaction **is**

**//** either a 'get' **or** a 'set' on the asset created by Init function**.** The Set

**//** method may create a new asset by specifying a new key**-**value pair**.**

func (t **\***SimpleAsset) Invoke(stub shim**.**ChaincodeStubInterface) peer**.**Response {

**//** Extract the function **and** args **from** the transaction proposal

fn, args :**=** stub**.**GetFunctionAndParameters()

var result string

var err error

**if** fn **==** "set" {

result, err **=** set(stub, args)

} **else** { **//** assume 'get' even **if** fn **is** nil

result, err **=** get(stub, args)

}

**if** err **!=** nil {

**return** shim**.**Error(err**.**Error())

}

**//** Return the result **as** success payload

**return** shim**.**Success([]byte(result))

}

**//** Set stores the asset (both key **and** value) on the ledger**.** If the key exists,

**//** it will override the value **with** the new one

func set(stub shim**.**ChaincodeStubInterface, args []string) (string, error) {

**if** len(args) **!=** 2 {

**return** "", fmt**.**Errorf("Incorrect arguments. Expecting a key and a value")

}

err :**=** stub**.**PutState(args[0], []byte(args[1]))

**if** err **!=** nil {

**return** "", fmt**.**Errorf("Failed to set asset: %s", args[0])

}

**return** args[1], nil

}

**//** Get returns the value of the specified asset key

func get(stub shim**.**ChaincodeStubInterface, args []string) (string, error) {

**if** len(args) **!=** 1 {

**return** "", fmt**.**Errorf("Incorrect arguments. Expecting a key")

}

value, err :**=** stub**.**GetState(args[0])

**if** err **!=** nil {

**return** "", fmt**.**Errorf("Failed to get asset: %s with error: %s", args[0], err)

}

**if** value **==** nil {

**return** "", fmt**.**Errorf("Asset not found: %s", args[0])

}

**return** string(value), nil

}

**//** main function starts up the chaincode **in** the container during instantiate

func main() {

**if** err :**=** shim**.**Start(new(SimpleAsset)); err **!=** nil {

fmt**.**Printf("Error starting SimpleAsset chaincode: %s", err)

}

}

### Building Chaincode

Now let’s compile your chaincode.

go get **-**u github**.**com**/**hyperledger**/**fabric**-**chaincode**-**go

go build

Assuming there are no errors, now we can proceed to the next step, testing your chaincode.

### Testing Using dev mode

Normally chaincodes are started and maintained by peer. However in “dev mode”, chaincode is built and started by the user. This mode is useful during chaincode development phase for rapid code/build/run/debug cycle turnaround.

We start “dev mode” by leveraging pre-generated orderer and channel artifacts for a sample dev network. As such, the user can immediately jump into the process of compiling chaincode and driving calls.

## Install Hyperledger Fabric Samples

If you haven’t already done so, please [Install Samples, Binaries and Docker Images](https://hyperledger-fabric.readthedocs.io/en/latest/install.html).

Navigate to the chaincode-docker-devmode directory of the fabric-samples clone:

cd chaincode**-**docker**-**devmode

Now open three terminals and navigate to your chaincode-docker-devmode directory in each.

## Terminal 1 - Start the network

docker**-**compose **-**f docker**-**compose**-**simple**.**yaml up

The above starts the network with the SingleSampleMSPSolo orderer profile and launches the peer in “dev mode”. It also launches two additional containers - one for the chaincode environment and a CLI to interact with the chaincode. The commands for create and join channel are embedded in the CLI container, so we can jump immediately to the chaincode calls.

* Note: the peer is not using TLS because the dev mode does not work with TLS.

## Terminal 2 - Build & start the chaincode

docker exec **-**it chaincode sh

You should see the following:

/opt/gopath/src/chaincode $

Now, compile your chaincode:

cd sacc

go build

Now run the chaincode:

CORE\_CHAINCODE\_ID\_NAME**=**mycc:0 CORE\_PEER\_TLS\_ENABLED**=**false **./**sacc **-**peer**.**address peer:7052

The chaincode is started with peer and chaincode logs indicating successful registration with the peer. Note that at this stage the chaincode is not associated with any channel. This is done in subsequent steps using the instantiate command.

## Terminal 3 - Use the chaincode

Even though you are in --peer-chaincodedev mode, you still have to install the chaincode so the life-cycle system chaincode can go through its checks normally. This requirement may be removed in future when in --peer-chaincodedev mode.

We’ll leverage the CLI container to drive these calls.

docker exec **-**it cli bash

peer chaincode install **-**p chaincodedev**/**chaincode**/**sacc **-**n mycc **-**v 0

peer chaincode instantiate **-**n mycc **-**v 0 **-**c '{"Args":["a","10"]}' **-**C myc

Now issue an invoke to change the value of “a” to “20”.

peer chaincode invoke **-**n mycc **-**c '{"Args":["set", "a", "20"]}' **-**C myc

Finally, query a. We should see a value of 20.

peer chaincode query **-**n mycc **-**c '{"Args":["query","a"]}' **-**C myc

## Testing new chaincode

By default, we mount only sacc. However, you can easily test different chaincodes by adding them to the chaincode subdirectory and relaunching your network. At this point they will be accessible in your chaincode container.

## Chaincode access control

Chaincode can utilize the client (submitter) certificate for access control decisions by calling the GetCreator() function. Additionally the Go shim provides extension APIs that extract client identity from the submitter’s certificate that can be used for access control decisions, whether that is based on client identity itself, or the org identity, or on a client identity attribute.

For example an asset that is represented as a key/value may include the client’s identity as part of the value (for example as a JSON attribute indicating that asset owner), and only this client may be authorized to make updates to the key/value in the future. The client identity library extension APIs can be used within chaincode to retrieve this submitter information to make such access control decisions.

See the [client identity (CID) library documentation](https://github.com/hyperledger/fabric-chaincode-go/blob/master/pkg/cid/README.md) for more details.

To add the client identity shim extension to your chaincode as a dependency, see [Managing external dependencies for chaincode written in Go](https://hyperledger-fabric.readthedocs.io/en/latest/chaincode4ade.html#vendoring).

## Managing external dependencies for chaincode written in Go

Your Go chaincode requires packages (like the chaincode shim) that are not part of the Go standard library. These packages must be included in your chaincode package.

There are [many tools available](https://github.com/golang/go/wiki/PackageManagementTools) for managing (or “vendoring”) these dependencies. The following demonstrates how to use govendor:

govendor init

govendor add **+**external **//** Add all external package, **or**

govendor add github**.**com**/**external**/**pkg **//** Add specific external package

This imports the external dependencies into a local vendor directory. If you are vendoring the Fabric shim or shim extensions, clone the Fabric repository to your $GOPATH/src/github.com/hyperledger directory, before executing the govendor commands.

Once dependencies are vendored in your chaincode directory, peer chaincode package and peer chaincode install operations will then include code associated with the dependencies into the chaincode package.

# Chaincode for Operators

## What is Chaincode?

Chaincode is a program, written in [Go](https://golang.org/), [Node.js](https://nodejs.org/), or [Java](https://java.com/en/) that implements a prescribed interface. Chaincode runs in a secured Docker container isolated from the endorsing peer process. Chaincode initializes and manages ledger state through transactions submitted by applications.

A chaincode typically handles business logic agreed to by members of the network, so it may be considered as a “smart contract”. Ledger updates created by a chaincode are scoped exclusively to that chaincode and can’t be accessed directly by another chaincode. However, within the same network, given the appropriate permission a chaincode may invoke another chaincode to access its state.

In the following sections, we will explore chaincode through the eyes of a blockchain network operator rather than an application developer. Chaincode operators can use this tutorial to learn how to use the Fabric chainode lifecycle to deploy and manage chaincode on their network.

## Chaincode lifecycle

The Fabric chaincode lifecycle is a process that allows multiple organizations to agree on how a chaincode will be operated before it can be used on a channel. The tutorial will discuss how a chaincode operator would use the Fabric lifecycle to perform the following tasks:

* [Install and define a chaincode](https://hyperledger-fabric.readthedocs.io/en/latest/chaincode4noah.html#install-and-define-a-chaincode)
* [Upgrade a chaincode](https://hyperledger-fabric.readthedocs.io/en/latest/chaincode4noah.html#upgrade-a-chaincode)
* [Deployment Scenarios](https://hyperledger-fabric.readthedocs.io/en/latest/chaincode4noah.html#deployment-scenarios)
* [Migrate to the new Fabric lifecycle](https://hyperledger-fabric.readthedocs.io/en/latest/chaincode4noah.html#migrate-to-the-new-fabric-lifecycle)

If you are upgrading from a v1.4.x network and need to edit your channel configurations to enable the new lifecycle, check out [Enabling the new chaincode lifecycle](https://hyperledger-fabric.readthedocs.io/en/latest/enable_cc_lifecycle.html).

## Install and define a chaincode

Fabric chaincode lifecycle requires that organizations agree to the parameters that define a chaincode, such as name, version, and the chaincode endorsement policy. Channel members come to agreement using the following four steps. Not every organization on a channel needs to complete each step.

1. **Package the chaincode:** This step can be completed by one organization or by each organization.
2. **Install the chaincode on your peers:** Every organization that will use the chaincode to endorse a transaction or query the ledger needs to complete this step.
3. **Approve a chaincode definition for your organization:** Every organization that will use the chaincode needs to complete this step. The chaincode definition needs to be approved by a sufficient number of organizations to satisfy the channel’s LifecycleEndorsment policy (a majority, by default) before the chaincode can be started on the channel.
4. **Commit the chaincode definition to the channel:** The commit transaction needs to be submitted by one organization once the required number of organizations on the channel have approved. The submitter first collects endorsements from enough peers of the organizations that have approved, and then submits the transaction to commit the chaincode definition.

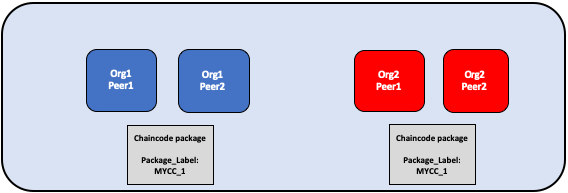
This tutorial provides a detailed overview of the operations of the Fabric chaincode lifecycle rather than the specific commands. To learn more about how to use the Fabric lifecycle using the Peer CLI, see [Install and define a chaincode](https://hyperledger-fabric.readthedocs.io/en/latest/build_network.html#install-define-chaincode) in the Building your First Network Tutorial or the [peer lifecycle command reference](https://hyperledger-fabric.readthedocs.io/en/latest/commands/peerlifecycle.html). To learn more about how to use the Fabric lifecycle using the Fabric SDK for Node.js, visit [How to install and start your chaincode](https://hyperledger.github.io/fabric-sdk-node/master/tutorial-chaincode-lifecycle.html).

### Step One: Packaging the smart contract

Chaincode needs to be packaged in a tar file before it can be installed on your peers. You can package a chaincode using the Fabric peer binaries, the Node Fabric SDK, or a third party tool such as GNU tar. When you create a chaincode package, you need to provide a chaincode package label to create a succinct and human readable description of the package.

If you use a third party tool to package the chaincode, the resulting file needs to be in the format below. The Fabric peer binaries and the Fabric SDKs will automatically create a file in this format.

* The chaincode needs to be packaged in a tar file, ending with a .tar.gz file extension.
* The tar file needs to contain two files (no directory): a metadata file “Chaincode-Package-Metadata.json” and another tar containing the chaincode files.
* “Chaincode-Package-Metadata.json” contains JSON that specifies the chaincode language, code path, and package label. You can see an example of a metadata file below:
* {"Path":"github.com/chaincode/fabcar/go","Type":"golang","Label":"fabcarv1"}

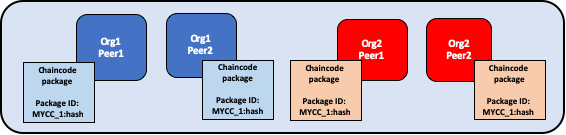


The chaincode is packaged separately by Org1 and Org2. Both organizations use MYCC\_1 as their package label in order to identify the package using the name and version. It is not necessary for organizations to use the same package label.

### Step Two: Install the chaincode on your peers

You need to install the chaincode package on every peer that will execute and endorse transactions. Whether using the CLI or an SDK, you need to complete this step using your **Peer Administrator**. Your peer will build the chaincode after the chaincode is installed, and return a build error if there is a problem with your chaincode. It is recommended that organizations only package a chaincode once, and then install the same package on every peer that belongs to their org. If a channel wants to ensure that each organization is running the same chaincode, one organization can package a chaincode and send it to other channel members out of band.

A successful install command will return a chaincode package identifier, which is the package label combined with a hash of the package. This package identifier is used to associate a chaincode package installed on your peers with a chaincode definition approved by your organization. **Save the identifier** for next step. You can also find the package identifier by querying the packages installed on your peer using the Peer CLI.



A peer administrator from Org1 and Org2 installs the chaincode package MYCC\_1 on the peers joined to the channel. Installing the chaincode package builds the chaincode and creates a package identifier of MYCC\_1:hash.

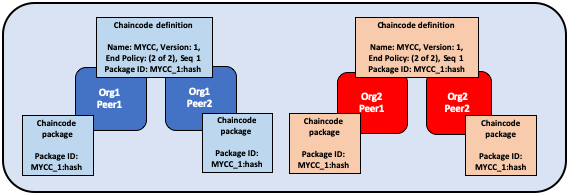
### Step Three: Approve a chaincode definition for your organization

The chaincode is governed by a **chaincode definition**. When channel members approve a chaincode definition, the approval acts as a vote by an organization on the chaincode parameters it accepts. These approved organization definitions allow channel members to agree on a chaincode before it can be used on a channel. The chaincode definition includes the following parameters, which need to be consistent across organizations:

* **Name:** The name that applications will use when invoking the chaincode.
* **Version:** A version number or value associated with a given chaincodes package. If you upgrade the chaincode binaries, you need to change your chaincode version as well.
* **Sequence:** The number of times the chaincode has been defined. This value is an integer, and is used to keep track of chaincode upgrades. For example, when you first install and approve a chaincode definition, the sequence number will be 1. When you next upgrade the chaincode, the sequence number will be incremented to 2.
* **Endorsement Policy:** Which organizations need to execute and validate the transaction output. The endorsement policy can be expressed as a string passed to the CLI or the SDK, or it can reference a policy in the channel config. By default, the endorsement policy is set to Channel/Application/Endorsement, which defaults to require that a majority of organizations in the channel endorse a transaction.
* **Collection Configuration:** The path to a private data collection definition file associated with your chaincode. For more information about private data collections, see the [Private Data architecture reference](https://hyperledger-fabric.readthedocs.io/en/master/private-data-arch.html).
* **Initialization:** All chaincode need to contain an Init function that is used to initialize the chaincode. By default, this function is never executed. However, you can use the chaincode definition to request that the Init function be callable. If execution of Init is requested, fabric will ensure that Init is invoked before any other function and is only invoked once.
* **ESCC/VSCC Plugins:** The name of a custom endorsement or validation plugin to be used by this chaincode.

The chaincode definition also includes the **Package Identifier**. This is a required parameter for each organization that wants to use the chaincode. The package ID does not need to be the same for all organizations. An organization can approve a chaincode definition without installing a chaincode package or including the identifier in the definition.

Each channel member that wants to use the chaincode needs to approve a chaincode definition for their organization. This approval needs to be submitted to the ordering service, after which it is distributed to all peers. This approval needs to be submitted by your **Organization Administrator**. After the approval transaction has been successfully submitted, the approved definition is stored in a collection that is available to all the peers of your organization. As a result you only need to approve a chaincode for your organization once, even if you have multiple peers.



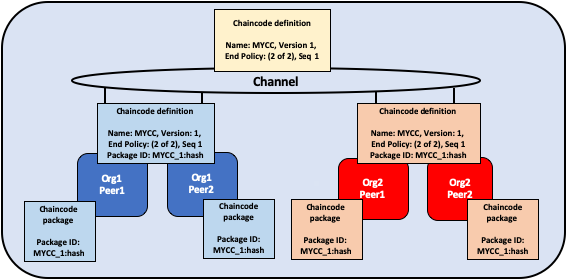
An organization administrator from Org1 and Org2 approve the chaincode definition of MYCC for their organization. The chaincode definition includes the chaincode name, version, and the endorsement policy, among other fields. Since both organizations will use the chaincode to endorse transactions, the approved definitions for both organizations need to include the packageID.

### Step Four: Commit the chaincode definition to the channel

Once a sufficient number of channel members have approved a chaincode definition, one organization can commit the definition to the channel. You can use the checkcommitreadiness command to check whether committing the chaincode definition should be successful based on which channel members have approved a definition before committing it to the channel using the peer CLI. The commit transaction proposal is first sent to the peers of channel members, who query the chaincode definition approved for their organizations and endorse the definition if their organization has approved it. The transaction is then submitted to the ordering service, which then commits the chaincode definition to the channel. The commit definition transaction needs to be submitted as the **Organization** **Administrator**.

The number of organizations that need to approve a definition before it can be successfully committed to the channel is governed by the Channel/Application/LifecycleEndorsement policy. By default, this policy requires that a majority of organizations in the channel endorse the transaction. The LifecycleEndorsement policy is separate from the chaincode endorsement policy. For example, even if a chaincode endorsement policy only requires signatures from one or two organizations, a majority of channel members still need to approve the chaincode definition according to the default policy. When committing a channel definition, you need to target enough peer organizations in the channel to satisfy your LifecycleEndorsement policy.

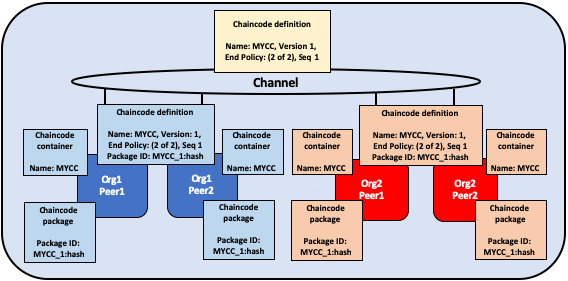
You can also set the Channel/Application/LifecycleEndorsement policy to be a signature policy and explicitly specify the set of organizations on the channel that can approve a chaincode definition. This allows you to create a channel where a select number of organizations act as chaincode administrators and govern the business logic used by the channel. You can also use a signature policy if your channel has a large number Idemix organizations, which cannot approve chaincode definitions or endorse chaincode and may prevent the channel from reaching a majority as a result.



One organization administrator from Org1 or Org2 commits the chaincode definition to the channel. The definition on the channel does not include the packageID.

An organization can approve a chaincode definition without installing the chaincode package. If an organization does not need to use the chaincode, they can approve a chaincode definition without a package identifier to ensure that the Lifecycle Endorsement policy is satisfied.

After the chaincode definition has been committed to the channel, the chaincode container will launch on all of the peers where the chaincode has been installed, allowing channel members to start using the chaincode. It may take a few minutes for the chaincode container to start. You can use the chaincode definition to require the invocation of the Init function to initialize the chaincode. If the invocation of the Init function is requested, the first invoke of the chaincode must be a call to the Init function. The invoke of the Init function is subject to the chaincode endorsement policy.

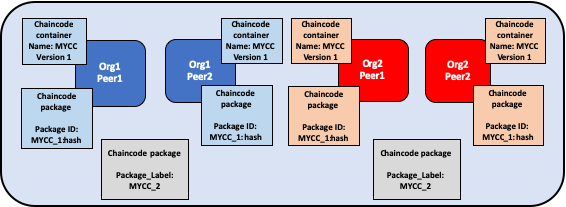


Once MYCC is defined on the channel, Org1 and Org2 can start using the chaincode. The first invoke of the chaincode on each peer starts the chaincode container on that peer.

## Upgrade a chaincode

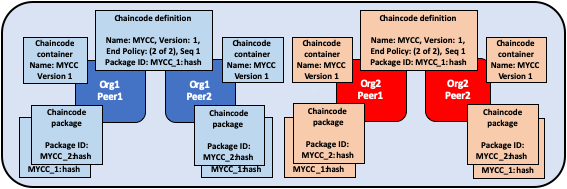
You can upgrade a chaincode using the same Fabric lifecycle process as you used to install and start the chainocode. You can upgrade the chaincode binaries, or only update the chaincode policies. Follow these steps to upgrade a chaincode:

1. **Repackage the chaincode:** You only need to complete this step if you are upgrading the chaincode binaries.



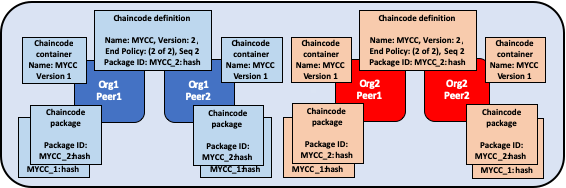
Org1 and Org2 upgrade the chaincode binaries and repackage the chaincode. Both organizations use a different package label.

1. **Install the new chaincode package on your peers:** Once again, you only need to complete this step if you are upgrading the chaincode binaries. Installing the new chaincode package will generate a package ID, which you will need to pass to the new chaincode definition. You also need to change the chaincode version, which is used by the lifecycle process to track if the chaincode binaries have been upgraded.



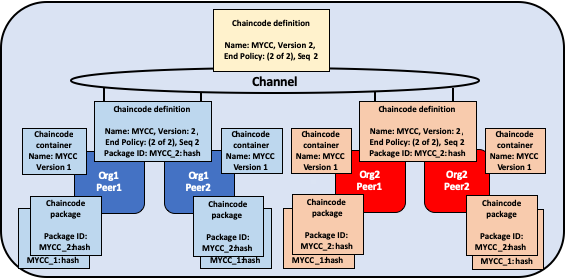
Org1 and Org2 install the new package on their peers. The installation creates a new packageID.

1. **Approve a new chaincode definition:** If you are upgrading the chaincode binaries, you need to update the chaincode version and the package ID in the chaincode definition. You can also update your chaincode endorsement policy without having to repackage your chaincode binaries. Channel members simply need to approve a definition with the new policy. The new definition needs to increment the **sequence** variable in the definition by one.



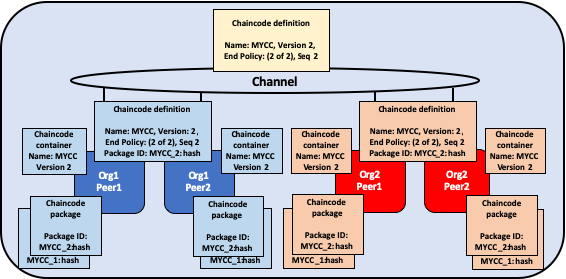
Organization administrators from Org1 and Org2 approve the new chaincode definition for their respective organizations. The new definition references the new packageID and changes the chaincode version. Since this is the first update of the chaincode, the sequence is incremented from one to two.

1. **Commit the definition to the channel:** When a sufficient number of channel members have approved the new chaincode definition, one organization can commit the new definition to upgrade the chaincode definition to the channel. There is no separate upgrade command as part of the lifecycle process.



An organization administrator from Org1 or Org2 commits the new chaincode definition to the channel.

After you commit the chaincode definition, a new chaincode container will launch with the code from the upgraded chaincode binaries. If you requested the execution of the Init function in the chaincode definition, you need to initialize the upgraded chaincode by invoking the Init function again after the new definition is successfully committed. If you updated the chaincode definition without changing the chaincode version, the chaincode container will remain the same and you do not need to invoke Init function.



Once the new definition has been committed to the channel, each peer will automatically start the new chaincode container.

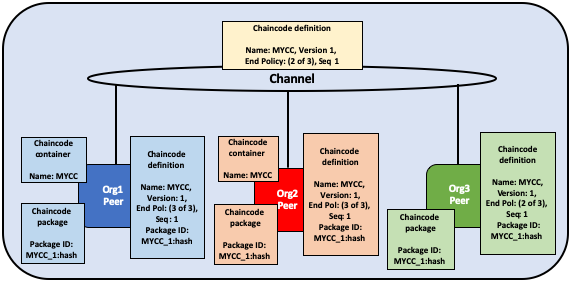
The Fabric chaincode lifecycle uses the **sequence** in the chaincode definition to keep track of upgrades. All channel members need to increment the sequence number by one and approve a new definition to upgrade the chaincode. The version parameter is used to track the chaincode binaries, and needs to be changed only when you upgrade the chaincode binaries.

## Deployment scenarios

The following examples illustrate how you can use the Fabric chaincode lifecycle to manage channels and chaincode.

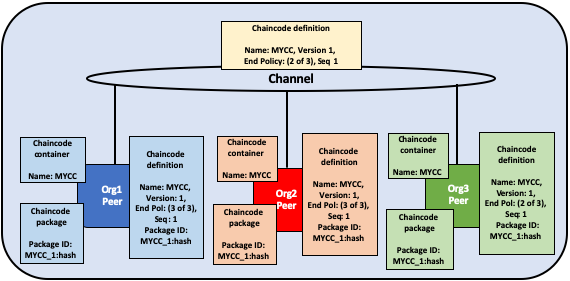
### Joining a channel

A new organization can join a channel with a chaincode already defined, and start using the chaincode after installing the chaincode package and approving the chaincode definition that has already been committed to the channel.



Org3 joins the channel and approves the same chaincode definition that was previously committed to the channel by Org1 and Org2.

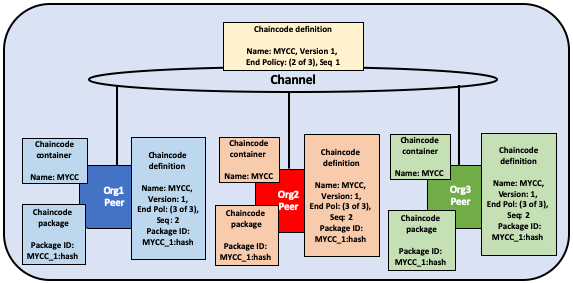
After approving the chaincode definition, the new organization can start using the chaincode after the package has been installed on their peers. The definition does not need to be committed again. If the endorsement policy is set the default policy that requires endorsements from a majority of channel members, then the endorsement policy will be updated automatically to include the new organization.



The chaincode container will start after the first invoke of the chaincode on the Org3 peer.

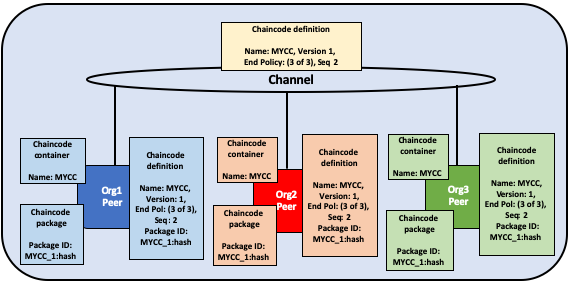
### Updating an endorsement policy

You can use the chaincode definition to update an endorsement policy without having to repackage or re-install the chaincode. Channel members can approve a chaincode definition with a new endorsement policy and commit it to the channel.



Org1, Org2, and Org3 approve a new endorsement policy requiring that all three organizations endorse a transaction. They increment the definition sequence from one to two, but do not need to update the chaincode version.

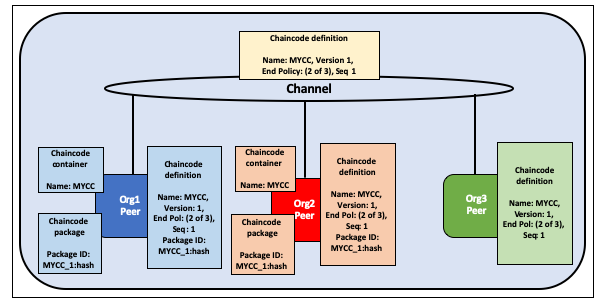
The new endorsement policy will take effect after the new definition is committed to the channel. Channel members do not have to restart the chaincode container by invoking the chaincode or executing the Init function in order to update the endorsement policy.



One organization commits the new chaincode definition to the channel to update the endorsement policy.

### Approving a definition without installing the chaincode

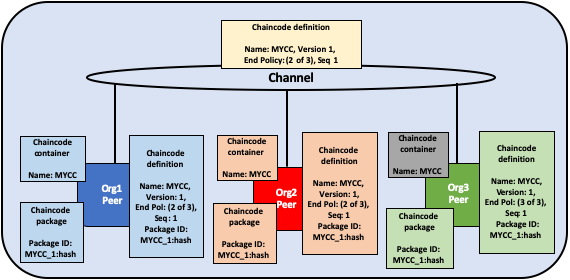
You can approve a chaincode definition without installing the chaincode package. This allows you to endorse a chaincode definition before it is committed to the channel, even if you do not want to use the chaincode to endorse transactions or query the ledger. You need to approve the same parameters as other members of the channel, but not need to include the packageID as part of the chaincode definition.



Org3 does not install the chaincode package. As a result, they do not need to provide a packageID as part of chaincode definition. However, Org3 can still endorse the definition of MYCC that has been committed to the channel.

### One organization disagrees on the chaincode definition

An organization that does not approve a chaincode definition that has been committed to the channel cannot use the chaincode. Organizations that have either not approved a chaincode definition, or approved a different chaincode definition will not be able to execute the chaincode on their peers.

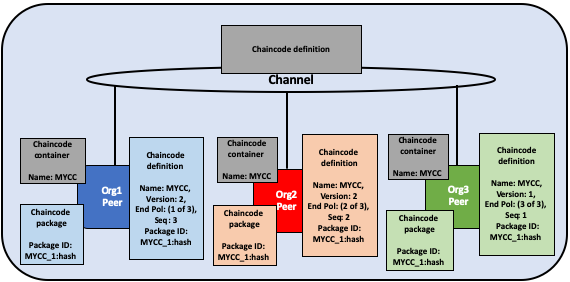


Org3 approves a chaincode definition with a different endorsement policy than Org1 and Org2. As a result, Org3 cannot use the MYCC chaincode on the channel. However, Org1 or Org2 can still get enough endorsements to commit the definition to the channel and use the chaincode. Transactions from the chaincode will still be added to the ledger and stored on the Org3 peer. However, the Org3 will not be able to endorse transactions.

An organization can approve a new chaincode definition with any sequence number or version. This allows you to approve the definition that has been committed to the channel and start using the chaincode. You can also approve a new chaincode definition in order to correct any mistakes made in the process of approving or packaging a chaincode.

### The channel does not agree on a chaincode definition

If the organizations on a channel do not agree on a chaincode definition, the definition cannot be committed to the channel. None of the channel members will be able to use the chaincode.

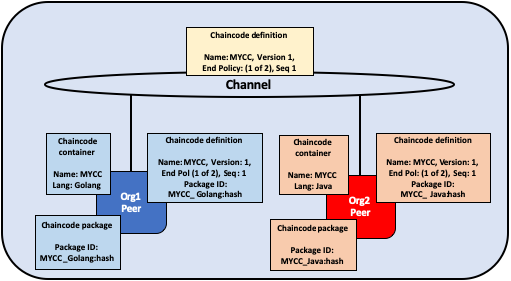


Org1, Org2, and Org3 all approve different chaincode definitions. As a result, no member of the channel can get enough endorsements to commit a chaincode definition to the channel. No channel member will be able to use the chaincode.

### Organizations install different chaincode packages

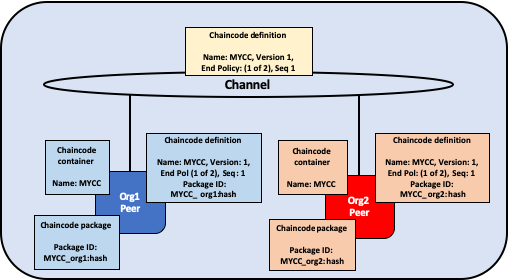
Each organization can use a different packageID when they approve a chaincode definition. This allows channel members to install different chaincode binaries that use the same endorsement policy and read and write to data in the same chaincode namespace.

Channel members can use this capability to install chaincode written in different languages and work with the language they are most comfortable. As long as the chaincode generates the same read-write sets, channel members using chaincode in different languages will be able to endorse transactions and commit them to the ledger. However, organizations should test that their chaincode is consistent and that they are able to generate valid endorsements before defining it on a channel in production.



Org1 installs a package of the MYCC chaincode written in Golang, while Org2 installs MYCC written in Java.

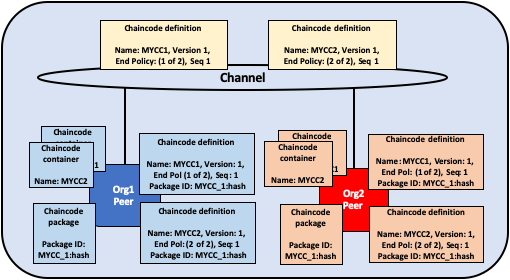
Organizations can also use this capability to install smart contracts that contain business logic that is specific to their organization. Each organization’s smart contract could contain additional validation that the organization requires before their peers endorse a transaction. Each organization can also write code that helps integrate the smart contract with data from their existing systems.



Org1 and Org2 each install versions of the MYCC chaincode containing business logic that is specific to their organization.

### Creating multiple chaincodes using one package

You can use one chaincode package to create multiple chaincode instances on a channel by approving and committing multiple chaincode definitions. Each definition needs to specify a different chaincode name. This allows you to run multiple instances of a smart contract on a channel, but have the contract be subject to different endorsement policies.



Org1 and Org2 use the MYCC\_1 chaincode package to approve and commit two different chaincode definitions. As a result, both peers have two chaincode containers running on their peers. MYCC1 has an endorsement policy of 1 out of 2, while MYCC2 has an endorsement policy of 2 out of 2.

## Migrate to the new Fabric lifecycle

For information about migrating to the new lifecycle, check out [Considerations for getting to v2.0](https://hyperledger-fabric.readthedocs.io/en/latest/upgrade_to_newest_version.html#chaincode-lifecycle).

If you need to update your channel configurations to enable the new lifecycle, check out [Enabling the new chaincode lifecycle](https://hyperledger-fabric.readthedocs.io/en/latest/enable_cc_lifecycle.html).