:doctype: book

include::attributes.txt[]

// Attributes [.topic] [#work-with] = Work with the {aws} CDK library :info\_titleabbrev: Work with the CDK library :keywords: {aws} CDK, IaC, Infrastructure as code, {aws} CloudFormation, {aws}, {aws} Cloud

== [abstract]

## Import and use the {aws} Cloud Development Kit ({aws} CDK) library to define your {aws} Cloud infrastructure with a xref:languages[supported programming language].

// Content start

Import and use the {aws} Cloud Development Kit ({aws} CDK) library to define your {aws} Cloud infrastructure with a xref:languages[supported programming language].

[#work-with-library] == Import the {aws} CDK Library

The xref:libraries[{aws} CDK Library] is often referred to by its TypeScript package name of aws-cdk-lib. The actual package name varies by language. The following is an example of how to install and import the CDK Library:

==== [role=“tablist”] TypeScript:: + [cols=“1,1”] |=== |*Install* |npm install aws-cdk-lib

|=== | *Import* | import \* as cdk from 'aws-cdk-lib'; |===

JavaScript:: + [cols=“1,1”] |=== |*Install* |npm install aws-cdk-lib

|=== | *Import* | const cdk = require('aws-cdk-lib'); |===

Python:: + [cols=“1,1”] |=== |*Install* |python -m pip install aws-cdk-lib

|=== | *Import* | import aws\_cdk as cdk |===

Java:: + [cols=“1,1”] |=== |*In pom.xml, add* |Group software.amazon.awscdk; artifact aws-cdk-lib

|=== | *Import* | import software.amazon.awscdk.App; |===

C#:: + [cols=“1,1”] |=== |*Install* |dotnet add package Amazon.CDK.Lib

|=== | *Import* | using Amazon.CDK; |===

Go:: + [cols=“1,1”] |=== |*Install* |go get github.com/aws/aws-cdk-go/awscdk/v2

|*Import* a|[source,go,subs=“verbatim,attributes”] — import ( “github.com/aws/aws-cdk-go/awscdk/v2” ) — |=== ====

The construct base class and supporting code is in the constructs library. Experimental constructs, where the API is still undergoing refinement, are distributed as separate modules.

[#work-with-library-reference] == Using the {aws} CDK API Reference

Use the xref:libraries-reference[{aws} CDK API reference] as you develop with the {aws} CDK.

Each module’s reference material is broken into the following sections.

* *Overview*: Introductory material you’ll need to know to work with the service in the {aws} CDK, including concepts and examples.
* *Constructs*: Library classes that represent one or more concrete {aws} resources. These are the “curated” (L2) resources or patterns (L3 resources) that provide a high-level interface with sane defaults.
* *Classes*: Non-construct classes that provide functionality used by constructs in the module.
* *Structs*: Data structures (attribute bundles) that define the structure of composite values such as properties (the props argument of constructs) and options.
* *Interfaces*: Interfaces, whose names all begin with “I”, define the absolute minimum functionality for the corresponding construct or other class. The CDK uses construct interfaces to represent {aws} resources that are defined outside your {aws} CDK app and referenced by methods such as Bucket.fromBucketArn().
* *Enums*: Collections of named values for use in specifying certain construct parameters. Using an enumerated value allows the CDK to check these values for validity during synthesis.
* *CloudFormation Resources*: These L1 constructs, whose names begin with “Cfn”, represent exactly the resources defined in the CloudFormation specification. They are automatically generated from that specification with each CDK release. Each L2 or L3 construct encapsulates one or more CloudFormation resources.
* *CloudFormation Property Types*: The collection of named values that define the properties for each CloudFormation Resource.

[#work-with-library-interfaces] == Interfaces compared with construct classes

The {aws} CDK uses interfaces in a specific way that may not be obvious even if you are familiar with interfaces as a programming concept.

The {aws} CDK supports using resources defined outside CDK applications using methods such as Bucket.fromBucketArn(). External resources cannot be modified and may not have all the functionality available with resources defined in your CDK app using e.g. the Bucket class. Interfaces, then, represent the bare minimum functionality available in the CDK for a given {aws} resource type, *including external resources*.

When instantiating resources in your CDK app, then, you should always use concrete classes such as Bucket. When specifying the type of an argument you are accepting in one of your own constructs, use the interface type such as IBucket if you are prepared to deal with external resources (that is, you won’t need to change them). If you require a CDK-defined construct, specify the most general type you can use.

Some interfaces are minimum versions of properties or options bundles associated with specific classes, rather than constructs. Such interfaces can be useful when subclassing to accept arguments that you’ll pass on to your parent class. If you require one or more additional properties, you’ll want to implement or derive from this interface, or from a more specific type.

= [NOTE]

Some programming languages supported by the {aws} CDK don’t have an interface feature. In these languages, interfaces are just ordinary classes. You can identify them by their names, which follow the pattern of an initial “I” followed by the name of some other construct (e.g. IBucket). The same rules apply.

====

[#work-with-cdk-dependencies] == Managing dependencies

Dependencies for your {aws} CDK app or library are managed using package management tools. These tools are commonly used with the programming languages.

Typically, the {aws} CDK supports the language’s standard or official package management tool if there is one. Otherwise, the {aws} CDK will support the language’s most popular or widely supported one. You may also be able to use other tools, especially if they work with the supported tools. However, official support for other tools is limited.

The {aws} CDK supports the following package managers:

[cols=“1,1”, options=“header”] |=== | Language | Supported package management tool

|=== | TypeScript/JavaScript | NPM (Node Package Manager) or Yarn |===

|=== | Python | PIP (Package Installer for Python) |===

|=== | Java | Maven |===

|=== | C# | NuGet |===

|=== | Go | Go modules |===

When you create a new project using the {aws} CDK CLI cdk init command, dependencies for the CDK core libraries and stable constructs are automatically specified.

For more information on managing dependencies for supported programming languages, see the following:

* xref:work-with-cdk-typescript-dependencies[Managing dependencies in TypeScript].
* xref:work-with-cdk-javascript-dependencies[Managing dependencies in JavaScript].
* xref:work-with-cdk-python-dependencies[Managing dependencies in Python].
* xref:work-with-cdk-java-dependencies[Managing dependencies in Java].
* xref:work-with-cdk-csharp-dependencies[Managing dependencies in C#].
* xref:work-with-cdk-go-dependencies[Managing dependencies in Go].

[#work-with-cdk-compare] == Comparing {aws} CDK in TypeScript with other languages

TypeScript was the first language supported for developing {aws} CDK applications. Therefore, a substantial amount of example CDK code is written in TypeScript. If you are developing in another language, it might be useful to compare how {aws} CDK code is implemented in TypeScript compared to your language of choice. This can help you use the examples throughout documentation.

[#work-with-cdk-compare-import] == Importing a module

==== [role=“tablist”] TypeScript/JavaScript:: TypeScript supports importing either an entire namespace, or individual objects from a namespace. Each namespace includes constructs and other classes for use with a given {aws} service. + [source,javascript,subs=“verbatim,attributes”] — // Import main CDK library as cdk import \* as cdk from ‘aws-cdk-lib’; // ES6 import preferred in TS const cdk = require(‘aws-cdk-lib’); // Node.js require() preferred in JS

// Import specific core CDK classes import { Stack, App } from ‘aws-cdk-lib’; const { Stack, App } = require(‘aws-cdk-lib’);

// Import {aws} S3 namespace as s3 into current namespace import { aws\_s3 as s3 } from ‘aws-cdk-lib’; // TypeScript const s3 = require(‘aws-cdk-lib/aws-s3’); // JavaScript

// Having imported cdk already as above, this is also valid const s3 = cdk.aws\_s3;

// Now use s3 to access the S3 types const bucket = s3.Bucket(…);

// Selective import of s3.Bucket import { Bucket } from ‘aws-cdk-lib/aws-s3’; // TypeScript const { Bucket } = require(‘aws-cdk-lib/aws-s3’); // JavaScript

// Now use Bucket to instantiate an S3 bucket const bucket = Bucket(…); —

Python:: Like TypeScript, Python supports namespaced module imports and selective imports. Namespaces in Python look like \*aws\_cdk.\*\*xxx\*, where *xxx* represents an {aws} service name, such as *s3* for Amazon S3. (Amazon S3 is used in these examples). + [source,python,subs=“verbatim,attributes”] —

= Import main CDK library as cdk

import aws\_cdk as cdk

= Selective import of specific core classes

from aws\_cdk import Stack, App

= Import entire module as s3 into current namespace

import aws\_cdk.aws\_s3 as s3

= s3 can now be used to access classes it contains

bucket = s3.Bucket(…)

= Selective import of s3.Bucket into current namespace

from aws\_cdk.s3 import Bucket

= Bucket can now be used to instantiate a bucket

## bucket = Bucket(…)

Java:: Java’s imports work differently from TypeScript’s. Each import statement imports either a single class name from a given package, or all classes defined in that package (using \\*). Classes may be accessed using either the class name by itself if it has been imported, or the *qualified* class name including its package. + Libraries are named like software.amazon.awscdk.services.xxx for the {aws} Construct Library (the main library is software.amazon.awscdk). The Maven group ID for {aws} CDK packages is software.amazon.awscdk. + [source,java,subs=“verbatim,attributes”] — // Make certain core classes available import software.amazon.awscdk.Stack; import software.amazon.awscdk.App;

// Make all Amazon S3 construct library classes available import software.amazon.awscdk.services.s3.\*;

// Make only Bucket and EventType classes available import software.amazon.awscdk.services.s3.Bucket; import software.amazon.awscdk.services.s3.EventType;

// An imported class may now be accessed using the simple class name (assuming that name // does not conflict with another class) Bucket bucket = Bucket.Builder.create(…).build();

// We can always use the qualified name of a class (including its package) even without an // import directive software.amazon.awscdk.services.s3.Bucket bucket = software.amazon.awscdk.services.s3.Bucket.Builder.create(…) .build();

// Java 10 or later can use var keyword to avoid typing the type twice var bucket = software.amazon.awscdk.services.s3.Bucket.Builder.create(…) .build(); —

C#:: In C#, you import types with the using directive. There are two styles. One gives you access to all the types in the specified namespace by using their plain names. With the other, you can refer to the namespace itself by using an alias. + Packages are named like +Amazon.CDK.{aws}.xxx+ for {aws} Construct Library packages. (The core module is Amazon.CDK.) + [source,csharp,subs=“verbatim,attributes”] — // Make CDK base classes available under cdk using cdk = Amazon.CDK;

// Make all Amazon S3 construct library classes available using Amazon.CDK.{aws}.S3;

// Now we can access any S3 type using its name var bucket = new Bucket(…);

// Import the S3 namespace under an alias using s3 = Amazon.CDK.{aws}.S3;

// Now we can access an S3 type through the namespace alias var bucket = new s3.Bucket(…);

// We can always use the qualified name of a type (including its namespace) even without a // using directive var bucket = new Amazon.CDK.{aws}.S3.Bucket(…); —

Go:: Each {aws} Construct Library module is provided as a Go package. + [source,go,subs=“verbatim,attributes”] — import ( “github.com/aws/aws-cdk-go/awscdk/v2” // CDK core package “github.com/aws/aws-cdk-go/awscdk/v2/awss3” // {aws} S3 construct library module )

// now instantiate a bucket bucket := awss3.NewBucket(…)

// use aliases for brevity/clarity import ( cdk “github.com/aws/aws-cdk-go/awscdk/v2” // CDK core package s3 “github.com/aws/aws-cdk-go/awscdk/v2/awss3” // {aws} S3 construct library module )

== bucket := s3.NewBucket(…)

====

[#work-with-cdk-compare-class] == Instantiating a construct

{aws} CDK construct classes have the same name in all supported languages. Most languages use the new keyword to instantiate a class (Python and Go do not). Also, in most languages, the keyword this refers to the current instance. (Python uses self by convention.) You should pass a reference to the current instance as the scope parameter to every construct you create.

The third argument to an {aws} CDK construct is props, an object containing attributes needed to build the construct. This argument may be optional, but when it is required, the supported languages handle it in idiomatic ways. The names of the attributes are also adapted to the language’s standard naming patterns.

==== [role=“tablist”] TypeScript/JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — // Instantiate default Bucket const bucket = new s3.Bucket(this, ‘amzn-s3-demo-bucket’);

// Instantiate Bucket with bucketName and versioned properties const bucket = new s3.Bucket(this, ‘amzn-s3-demo-bucket’, { bucketName: ‘amzn-s3-demo-bucket’, versioned: true, });

// Instantiate Bucket with websiteRedirect, which has its own sub-properties const bucket = new s3.Bucket(this, ‘amzn-s3-demo-bucket’, { websiteRedirect: {host: ‘aws.amazon.com’}}); —

Python:: Python doesn’t use a new keyword when instantiating a class. The properties argument is represented using keyword arguments, and the arguments are named using snake\_case. + If a props value is itself a bundle of attributes, it is represented by a class named after the property, which accepts keyword arguments for the subproperties. + In Python, the current instance is passed to methods as the first argument, which is named self by convention. + [source,python,subs=“verbatim,attributes”] —

= Instantiate default Bucket

bucket = s3.Bucket(self, “amzn-s3-demo-bucket”)

= Instantiate Bucket with bucket\_name and versioned properties

bucket = s3.Bucket(self, “amzn-s3-demo-bucket”, bucket\_name=“amzn-s3-demo-bucket”, versioned=true)

= Instantiate Bucket with website\_redirect, which has its own sub-properties

bucket = s3.Bucket(self, “amzn-s3-demo-bucket”, website\_redirect=s3.WebsiteRedirect( host\_name=“aws.amazon.com”)) —

Java:: In Java, the props argument is represented by a class named XxxxProps (for example, BucketProps for the Bucket construct’s props). You build the props argument using a builder pattern. + Each XxxxProps class has a builder. There is also a convenient builder for each construct that builds the props and the construct in one step, as shown in the following example. + Props are named the same as in TypeScript, using camelCase. + [source,java,subs=“verbatim,attributes”] — // Instantiate default Bucket Bucket bucket = Bucket(self, “amzn-s3-demo-bucket”);

// Instantiate Bucket with bucketName and versioned properties Bucket bucket = Bucket.Builder.create(self, “amzn-s3-demo-bucket”) .bucketName(“amzn-s3-demo-bucket”).versioned(true) .build();

= Instantiate Bucket with websiteRedirect, which has its own sub-properties

Bucket bucket = Bucket.Builder.create(self, “amzn-s3-demo-bucket”) .websiteRedirect(new websiteRedirect.Builder() .hostName(“aws.amazon.com”).build()) .build(); —

C#:: In C#, props are specified using an object initializer to a class named XxxxProps (for example, BucketProps for the Bucket construct’s props). + Props are named similarly to TypeScript, except using PascalCase. + It is convenient to use the var keyword when instantiating a construct, so you don’t need to type the class name twice. However, your local code style guide may vary. + [source,csharp,subs=“verbatim,attributes”] — // Instantiate default Bucket var bucket = Bucket(self, “amzn-s3-demo-bucket”);

// Instantiate Bucket with BucketName and Versioned properties var bucket = Bucket(self, “amzn-s3-demo-bucket”, new BucketProps { BucketName = “amzn-s3-demo-bucket”, Versioned = true});

// Instantiate Bucket with WebsiteRedirect, which has its own sub-properties var bucket = Bucket(self, “amzn-s3-demo-bucket”, new BucketProps { WebsiteRedirect = new WebsiteRedirect { HostName = “aws.amazon.com” }}); —

Go:: To create a construct in Go, call the function NewXxxxxx where Xxxxxxx is the name of the construct. The constructs’ properties are defined as a struct. + In Go, all construct parameters are pointers, including values like numbers, Booleans, and strings. Use the convenience functions like jsii.String to create these pointers. + [source,go,subs=“verbatim,attributes”] — // Instantiate default Bucket bucket := awss3.NewBucket(stack, jsii.String(“amzn-s3-demo-bucket”), nil)

…. // Instantiate Bucket with BucketName and Versioned properties bucket1 := awss3.NewBucket(stack, jsii.String(“amzn-s3-demo-bucket”), &awss3.BucketProps{ BucketName: jsii.String(“amzn-s3-demo-bucket”), Versioned: jsii.Bool(true), })

// Instantiate Bucket with WebsiteRedirect, which has its own sub-properties bucket2 := awss3.NewBucket(stack, jsii.String(“amzn-s3-demo-bucket”), &awss3.BucketProps{ WebsiteRedirect: &awss3.RedirectTarget{ HostName: jsii.String(“aws.amazon.com”), }}) —- ==== ….

[#work-with-cdk-compare-members] == Accessing members

It is common to refer to attributes or properties of constructs and other {aws} CDK classes and use these values as, for example, inputs to build other constructs. The naming differences described previously for methods apply here also. Furthermore, in Java, it is not possible to access members directly. Instead, a getter method is provided.

==== [role=“tablist”] TypeScript/JavaScript:: Names are camelCase. + [source,javascript,subs=“verbatim,attributes”] — bucket.bucketArn —

Python:: Names are snake\_case. + [source,python,subs=“verbatim,attributes”] — bucket.bucket\_arn —

Java:: A getter method is provided for each property; these names are camelCase. + [source,java,subs=“verbatim,attributes”] — bucket.getBucketArn() —

C#:: Names are PascalCase. + [source,javascript,subs=“verbatim,attributes”] — bucket.BucketArn —

Go:: Names are PascalCase. + [source,javascript,subs=“verbatim,attributes”] — bucket.BucketArn — ====

[#work-with-cdk-compare-enums] == Enum constants

Enum constants are scoped to a class, and have uppercase names with underscores in all languages (sometimes referred to as SCREAMING\_SNAKE\_CASE). Since class names also use the same casing in all supported languages except Go, qualified enum names are also the same in these languages.

== [source,javascript,subs=“verbatim,attributes”]

## s3.BucketEncryption.KMS\_MANAGED

In Go, enum constants are attributes of the module namespace and are written as follows.

== [source,javascript,subs=“verbatim,attributes”]

## awss3.BucketEncryption\_KMS\_MANAGED

[#work-with-cdk-compare-object] == Object interfaces

The {aws} CDK uses TypeScript object interfaces to indicate that a class implements an expected set of methods and properties. You can recognize an object interface because its name starts with I. A concrete class indicates the interfaces that it implements by using the implements keyword.

==== [role=“tablist”] TypeScript/JavaScript:: + NOTE: JavaScript doesn’t have an interface feature. You can ignore the implements keyword and the class names following it. +

== [source,javascript,subs=“verbatim,attributes”]

import { IAspect, IConstruct } from ‘aws-cdk-lib’;

class MyAspect implements IAspect { public visit(node: IConstruct) { console.log(‘Visited’, node.node.path); } } —

Python:: Python doesn’t have an interface feature. However, for the {aws} CDK you can indicate interface implementation by decorating your class with @jsii.implements(interface). + [source,python,subs=“verbatim,attributes”] — from aws\_cdk import IAspect, IConstruct import jsii

@jsii.implements(IAspect) class MyAspect(): def visit(self, node: IConstruct) -> None: print(“Visited”, node.node.path) —

Java:: + [source,java,subs=“verbatim,attributes”] — import software.amazon.awscdk.IAspect; import software.amazon.awscdk.IConstruct;

public class MyAspect implements IAspect { public void visit(IConstruct node) { System.out.format(“Visited %s”, node.getNode().getPath()); } } —

C#:: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK;

public class MyAspect : IAspect { public void Visit(IConstruct node) { System.Console.WriteLine($“Visited ${node.Node.Path}”); } } —

Go:: Go structs do not need to explicitly declare which interfaces they implement. The Go compiler determines implementation based on the methods and properties available on the structure. For example, in the following code, MyAspect implements the IAspect interface because it provides a Visit method that takes a construct. + [source,go,subs=“verbatim,attributes”] — type MyAspect struct { }

func (a MyAspect) Visit(node constructs.IConstruct) { fmt.Println(“Visited”, \*node.Node().Path()) } — ====

include::work-with-cdk-typescript.adoc[leveloffset=+1]

include::work-with-cdk-javascript.adoc[leveloffset=+1]

include::work-with-cdk-python.adoc[leveloffset=+1]

include::work-with-cdk-java.adoc[leveloffset=+1]

include::work-with-cdk-csharp.adoc[leveloffset=+1]

include::work-with-cdk-go.adoc[leveloffset=+1]