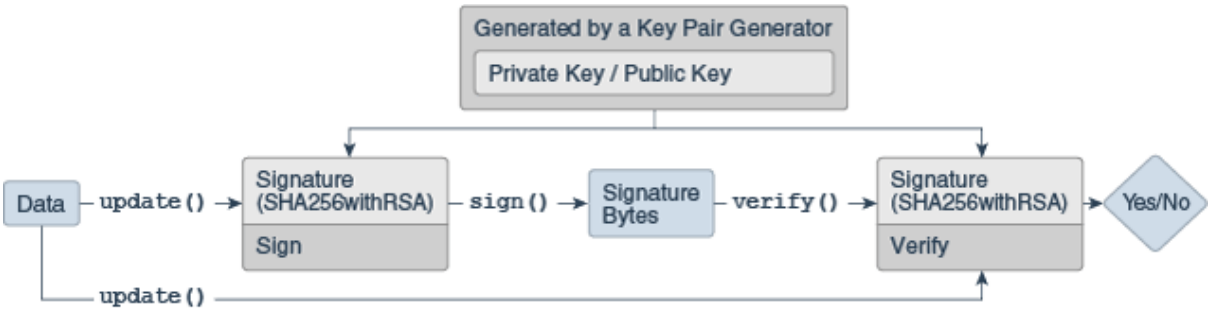
The `Signature` Class

The `Signature` class is an engine class designed to provide the functionality of a cryptographic digital signature algorithm such as DSA or RSAwithMD5. A cryptographically secure signature algorithm takes arbitrary-sized input and a private key and generates a relatively short (often fixed-size) string of bytes, called the *signature*, with the following properties:

- Only the owner of a private/public key pair is able to create a signature. It should be computationally infeasible for anyone having a public key to recover the private key.
- Given the public key corresponding to the private key used to generate the signature, it should be possible to verify the authenticity and integrity of the input.
- The signature and the public key do not reveal anything about the private key.

It can also be used to verify whether or not an alleged signature is in fact the authentic signature of the data associated with it.



Description of Figure The Signature Class

A `Signature` object is initialized for signing with a Private Key and is given the data to be signed. The resulting signature bytes are typically kept with the signed data. When verification is needed, another `Signature` object is created and initialized for verification and given the corresponding Public Key. The data and the signature bytes are fed to the signature object, and if the data and signature match, the `Signature` object reports success.

Even though a signature seems similar to a message digest, they have very different purposes in the type of protection they provide. In fact, algorithms such as "SHA256withRSA" use the message digest "SHA256" to initially "compress" the large data sets into a more manageable form, then sign the resulting 32 byte message digest with the "RSA" algorithm.

Please see the Examples section for an example of signing and verifying data.

`Signature` Object States

`Signature` objects are modal objects. This means that a `Signature` object is always in a given state, where it may only do one type of operation. States are represented as final integer constants defined in their respective classes.

The three states a `Signature` object may have are:

- `UNINITIALIZED`
- `SIGN`
- `VERIFY`

When it is first created, a `Signature` object is in the `UNINITIALIZED` state. The `Signature` class defines two initialization methods, `initSign` and `initVerify`, which change the state to `SIGN` and `VERIFY`, respectively.

Creating a `Signature` Object

The first step for signing or verifying a signature is to create a `Signature` instance. `Signature` objects are obtained by using one of the `Signature` `getInstance()` static factory methods.

Initializing a `Signature` Object

A `Signature` object must be initialized before it is used. The initialization method depends on whether the object is going to be used for signing or for verification.

If it is going to be used for signing, the object must first be initialized with the private key of the entity whose signature is going to be generated. This initialization is done by calling the method:

```
final void initSign(PrivateKey privateKey)
```

This method puts the `Signature` object in the `SIGN` state.
If instead the `Signature` object is going to be used for verification, it must first be initialized with the public key of the entity whose signature is going to be verified. This initialization is done by calling either of these methods:

```
   final void initVerify(PublicKey publicKey)

    final void initVerify(Certificate certificate)
```

This method puts the `Signature` object in the `VERIFY` state.

Signing

If the `Signature` object has been initialized for signing (if it is in the `SIGN` state), the data to be signed can then be supplied to the object. This is done by making one or more calls to one of the `update` methods:

```
final void update(byte b)
final void update(byte[] data)
final void update(byte[] data, int off, int len)
```

Calls to the `update` method(s) should be made until all the data to be signed has been supplied to the `Signature` object.

To generate the signature, simply call one of the `sign` methods:

```
final byte[] sign()
final int sign(byte[] outbuf, int offset, int len)
```

The first method returns the signature result in a byte array. The second stores the signature result in the provided buffer *outbuf*, starting at *offset*. *len* is the number of bytes in *outbuf* allotted for the signature. The method returns the number of bytes actually stored.

Signature encoding is algorithm specific. See the Standard Names document for more information about the use of ASN.1 encoding in the Java Cryptography Architecture.

A call to a `sign` method resets the signature object to the state it was in when previously initialized for signing via a call to `initSign`. That is, the object is reset and available to generate another signature with the same private key, if desired, via new calls to `update` and `sign`.

Alternatively, a new call can be made to `initSign` specifying a different private key, or to `initVerify` (to initialize the `Signature` object to verify a signature).

Verifying

If the `Signature` object has been initialized for verification (if it is in the `VERIFY` state), it can then verify if an alleged signature is in fact the authentic signature of the data associated with it. To start the process, the data to be verified (as opposed to the signature itself) is supplied to the object. The data is passed to the object by calling one of the `update` methods:

```
final void update(byte b)
final void update(byte[] data)
final void update(byte[] data, int off, int len)
```

Calls to the `update` method(s) should be made until all the data to be verified has been supplied to the `Signature` object. The signature can now be verified by calling one of the `verify` methods:

```
final boolean verify(byte[] signature)

final boolean verify(byte[] signature, int offset, int length)
```

The argument must be a byte array containing the signature. This byte array would hold the signature bytes which were returned by a previous call to one of the `sign` methods.

The `verify` method returns a `boolean` indicating whether or not the encoded signature is the authentic signature of the data supplied to the `update` method(s).

A call to the `verify` method resets the signature object to its state when it was initialized for verification via a call to `initVerify`. That is, the object is reset and available to verify another signature from the identity whose public key was specified in the call to `initVerify`.

Alternatively, a new call can be made to `initVerify` specifying a different public key (to initialize the `Signature` object for verifying a signature from a different entity), or to `initSign` (to initialize the `Signature` object for generating a signature).