I’m sure you’ve heard of public key encryption (PKE). You probably use it every day—if not in your code or on your servers, then on the Internet. When you use secure socket layers (SSL) in your browser or when you see the lock or key symbol and see https:\\ (with the s) as the protocol in the address bar, you are using PKE and other encryption. SSL is used to send private data such as your credit card number in encrypted form to online merchants.

The basic concept of PKE is that two entities share a set of two keys (one public and one private) and use them to encrypt data for and decrypt data from each other. The keys are tied to one another so that anything encrypted with one of the pair (e.g., the private key) can only be decrypted with the other (e.g., the public key). This encryption/decryption works in both directions.

The most important aspect of PKE is that one of the keys is private. The private key is never disclosed to anyone; however, the public key can be given to any and all requestors. How does this help privacy?

The one requesting the public key can be sure that if that key successfully decrypts data, then it came from the entity with the private key. Only the private key can encrypt data that can be decrypted with the public key.

The converse is also true: only the public key can encrypt data that can be decrypted by the private key, so the one using the public key can be sure that only the entity with the private key can decrypt data that they encrypted.

Because the public key is, well, public, anyone can decrypt data sent from the originator. Also anyone can encrypt data and send it to the originator for decryption. This is not a flaw, but it is an aspect of PKE that we need to be aware of and account for.

Generate Keys on the Client

We will have the client computer generate a set of keys. That computer will send artifacts (components) of the public key to the Oracle database so that Oracle can build a copy of the public key. Then the Oracle database can encrypt data using the public key that only the originating client can decrypt.

This approach may sound like a complete solution, but there are a couple concerns that we will not address until we get to the next chapter’s discussion on secret password encryption. First, anyone can read the public key artifacts as they traverse the network (that is, anyone with software to read all packets going across the network, like a sniffer.) That means that we have to assume that everyone has the public key and that everyone sees and can decrypt data (if any) that the client encrypts with the private key and sends to the server.

The second concern is that PKE, at least the version we are using, does not lend itself to encrypting large amounts data. It is a block cipher with a limited block size. For example, if the block size for our PKE keys is limited to 128 bytes, we would have to break the data into portions of that size and encrypt each portion individually. On the other side of this transaction, the recipient would have to decrypt each portion and reassemble the original data.
To handle larger amounts of data, there are a couple of methods: cipher block chaining (CBC) and stream encryption. With CBC, large data is broken up into appropriate block sizes for encryption and then decrypted and reassembled automatically for the user. (Whew, that takes a lot of burden off our shoulders.) With stream encryption, each bit, byte, or block of bytes would be encrypted/decrypted as it passed through the stream. A stream is simply a channel for data en route. You put bytes of data into a stream, and take bytes of data out in the same order: first in first out (fifo). A stream can exist when reading/writing data to storage, or across the network, or simply from one place (structure) in memory to another.

**RSA Public Key Cryptography**

We will be using RSA public key cryptography for our PKE encryption algorithm. RSA stands for the last names of the creators of the algorithm: Rivest, Shamir, and Adleman.

Because RSA uses a different key for encryption (e.g., private) from what is required for decryption (e.g., public), it is called an asymmetric algorithm. All PKE is asymmetric encryption. With a long key length, RSA is a very trustworthy encryption algorithm.

**Java Code to Generate and Use RSA Keys**

All our code for accomplishing Oracle database and Java security will reside in a single Java class (there are some small exceptions; we will have some separate Java classes for testing our processes). As we walk through the remaining chapters of this book, we are going to develop security code in phases, adding layers and concepts as we progress. Our single Java class will grow over time.

Our class will be called `OracleJavaSecure`, and we will define it in a package called `orajavsec`. Because we do not have a single version of this file, we are going to have multiple directories (one per chapter) where different versions of this Java code reside. This will make compiling and running a bit more difficult, but I will provide instructions as needed to reference these files.

> **Note** You can find the following code in the file `Chapter5/orajavsec/OracleJavaSecure.java`. I recommend that you open that file and refer to it as we proceed through this chapter.

---

Creating a Set of Keys

Listing 5-1 shows the code that is used for creating a set of PKE keys. This code, along with other Java code in this chapter, comes from the `OracleJavaSecure` class.

**Listing 5-1. Create PKE Keys, `makeLocRSAKeys()`**

```java
private static SecureRandom random;
private static int keyLengthRSA = 1024;
private static Key locRSAPrivKey;
private static RSAPublicKey locRSAPubKey;
private static void makeLocRSAKeys() throws Exception {
    random = new SecureRandom();
    KeyPairGenerator generator = KeyPairGenerator.getInstance( "RSA" );
```