Elliptic Curve Key Pair Generation and Key Factories

Key pair generation in elliptic curve follows the same principles as the other algorithms, the main difference being that, unlike algorithms such as RSA, elliptic curve keys exist only in the context of a particular elliptic curve and require to have curve parameters associated with them to be of any use.

Having said that, there is one anomaly with elliptic curve over other algorithms in that there are two APIs supported by the provider for using them. The reason for this is that JDK elliptic curve support was only introduced with the release of JDK 1.5. Prior to that providers supporting elliptic curve had to include some provider specific classes to allow it to be used, and as Bouncy Castle has supported elliptic curve since release 1.04 it had to provide it's own API.

Other than differences in parameters the generation of elliptic curve keys is identical for both Fp and F2m.

Using ASN.1 Encoding

Like other asymmetric algorithms, elliptic curve private keys produce DER encodings of PKCS8 PrivateKeyInfo objects and elliptic curve public keys produce DER encodings of X.509 SubjectPublicKeyInfo objects.

The following example shows a simple case of copying a key pair using the `getEncoded()` method on the public and private keys and the `X509EncodedKeySpec` and `PKCS8EncodedKeySpec` classes.

```
import java.security.KeyFactory;
import java.security.spec.X509EncodedKeySpec;
import java.security.spec.PKCS8EncodedKeySpec;
...
KeyPair  pair = ...;
KeyFactory fact = KeyFactory.getInstance("ECDSA", "BC");
PublicKey public = fact.generatePublic(new
X509EncodedKeySpec(pair.getPublic().getEncoded()));
PrivateKey private = fact.generatePrivate(new
PKCS8EncodedKeySpec(pair.getPrivate().getEncoded()));
```

Using the Bouncy Castle Specific APIs

The Bouncy Castle API for elliptic curve consists of a collection of interfaces and classes defined in org.bouncycastle.jce, org.bouncycastle.jce.interfaces, and org.bouncycastle.jce.spec packages which provide provider specific support for elliptic curve keys, parameters, and named curve handling.

Key Pair Generation

Key pair generation can be done using explicitly created parameters or by retrieving a named curve from a lookup table.

From Explicit Parameters

An org.bouncycastle.jce.ECParameterSpec is required to construct an elliptic curve key. The long way of creating one of these is to create the ECParameterSpec object from a Bouncy Castle ECCurve object and an associated base point and order.

Normally you'd only do this if the curve you want is not already present in one of the named curve tables (see below), but if you had a set of parameters you wanted to use it would look something like this:

```
import org.bouncycastle.math.ec.ECCurve;
import org.bouncycastle.jce.spec.ECParameterSpec;
...
ECCurve curve = new ECCurve.Fp(
            new
BigInteger("8834235323891921647916487503603088853144765972529603627924508606096998
9"), // q
            new
BigInteger("7fffffffffffffffffffffff7fffffffffff8000000000007ffffffffc", 16), //
a
            new
BigInteger("6b016c3bdcf18941d0d654921475ca71a9db2fb27d1d37796185c2942c0a", 16)); //
b
ECParameterSpec ecSpec = new ECParameterSpec(
            curve,

curve.decodePoint(Hex.decode("020ffa963cdca8816ccc33b8642bedf905c3d358573d3f27fbbd3
b3cb9aaaf")), // G
            new
BigInteger("8834235323891921647916487503603088480755034169162775227534542470280730
7")); // n
KeyPairGenerator g = KeyPairGenerator.getInstance("ECDSA", "BC");
g.initialize(ecSpec, new SecureRandom());
KeyPair pair = g.generateKeyPair();
```

As you can see it is a two step process. First you need to create the curve and then you need to associate the curve with a base point and an order using an ECParameterSpec which is then used to initialise the KeyPairGenerator object.

From Named Curves

Named curves are handled in the Bouncy Castle provider by associating a parameter set with a name using an extension of ECParameterSpec, ECNamedCurveParameterSpec, which can be found in org.bouncycastle.jce.spec. Normally you would not create one of these parameter spec objects directly, but you would retrieve it from one of the two lookup tables in org.bouncycastle.jce - ECNamedCurveTable if you are using ECDSA, or ECGOST3410NamedCurveTable if you are using GOST310-2001. Both classes support a getNames() method which will tell you what named curves are currently supported.

Assuming you were wanting to use the X9.62 curve prime192v1, the code would look something like this:

```
import org.bouncycastle.jce.spec.ECParameterSpec;
...
ECParameterSpec ecSpec = ECNamedCurveTable.getParameterSpec("prime192v1");
KeyPairGenerator g = KeyPairGenerator.getInstance("ECDSA", "BC");
g.initialize(ecSpec, new SecureRandom());
KeyPair pair = g.generateKeyPair();
```

Using a KeyFactory

From Explicit Parameters

The Bouncy Castle provider also supports key spec objects for cases where the key material is already available and the use of a KeyPairGenerator is not required. In this case the regular KeyFactory class is used and the Bouncy Castle specific classes ECPublicKeySpec and ECPrivateKeySpec are used to hold the material for the public and private keys respectively.

```
import org.bouncycastle.math.ec.ECCurve;
import org.bouncycastle.jce.spec.ECParameterSpec;
import org.bouncycastle.jce.spec.ECPrivateKeySpec;
import org.bouncycastle.jce.spec.ECPublicKeySpec;
...
ECCurve curve = new ECCurve.F2m(
            239, // m
            36, // k
            new
BigInteger("32010857077C5431123A46B808906756F543423E8D27877578125778AC76", 16), //
a
            new
BigInteger("790408F2EEDAF392B012EDEFB3392F30F4327C0CA3F31FC383C422AA8C16", 16)); //
b
ECParameterSpec params = new ECParameterSpec(
            curve,

curve.decodePoint(Hex.decode("0457927098FA932E7C0A96D3FD5B706EF7E5F5C156E16B7E7C860
38552E91D61D8EE5077C33FECF6F1A16B268DE469C3C7744EA9A971649FC7A9616305")), // G
            new
BigInteger("2208558830972980411979121875928648145578869377671323093671504120741178
3"), // n
            BigInteger.valueOf(4)); // h
ECPrivateKeySpec priKeySpec = new ECPrivateKeySpec(
            new
BigInteger("1456427555219115346513212300075341203043918714616464614664646674949479 9
0"), // d
            params);
ECPublicKeySpec pubKeySpec = new ECPublicKeySpec(

curve.decodePoint(Hex.decode("045894609CCECF9A92533F630DE713A958E96C97CCB8F5ABB5A68
8A238DEED6DC2D9D0C94EBFB7D526BA6A61764175B99CB6011E2047F9F067293F57F5")), // Q
            params);
```

As you can see the first step is identical to that used for the KeyGenerator, except this time the ECParameterSpec is used to create an ECPrivateKeySpec containing the private value and the parameters, and an ECPublicKeySpec containing the public point and the curve parameters.

These can then be passed to a KeyFactory as follows:

```
PrivateKey          sKey = f.generatePrivate(priKeySpec);
PublicKey           vKey = f.generatePublic(pubKeySpec);
```

and the resulting keys can then be used as the ones produced by the KeyPairGenerator were.

With Named Curves

As with the key pair generation example, if you know the curve associated with the keys you have been given is for a named curve, you can replace the construction of the ECParmeterSpec above with a named curve lookup using one of the named curve tables from org.bouncycastle.jce.

Using the JDK APIs

If you are using JDK 1.5 or later there is local support in the JDK for generation of elliptic curve keys.

Key Pair Generation

With Explicit Parameters

If you're using explicit parameters to generate keys:

```
import java.security.spec.ECParameterSpec;
import java.security.spec.EllipticCurve;
....
EllipticCurve curve = new EllipticCurve(
            new ECFieldFp(new
BigInteger("883423532389192164791648750360308885314476597252960362792450860609699839")), // q
            new
BigInteger("7fffffffffffffffffffff7fffffffffff8000000000007ffffffffffc", 16), // a
            new
BigInteger("6b016c3bdcf18941d0d654921475ca71a9db2fb27d1d37796185c2942c0a", 16)); // b
ECParameterSpec ecSpec = new ECParameterSpec(
            curve,
            ECPointUtil.decodePoint(curve,
Hex.decode("020ffa963cdca8816ccc33b8642bedf905c3d358573d3f27fbbd3b3cb9aaaf")), // G
            new
BigInteger("883423532389192164791648750360308884807550341691627752275345424702807307"), // n
            1); // h
KeyPairGenerator g = KeyPairGenerator.getInstance("ECDSA", "BC");
g.initialize(ecSpec, new SecureRandom());
KeyPair pair = g.generateKeyPair();
```

With Named Curves

The JDK also supports the use of Named Curves using the ECGenParameterSpec, which simply passes the name of the curve to the provider for interpretation. For example to use the X9.62 curve prime192v1 with the Bouncy Castle provider to generate an Elliptic Curve key pair the code would look something like the following:

```
import java.security.spec.ECGenParameterSpec;
...
ECGenParameterSpec     ecGenSpec = new ECGenParameterSpec("prime192v1");
KeyPairGenerator    g = KeyPairGenerator.getInstance("ECDSA", "BC");
g.initialize(ecGenSpec, new SecureRandom());
KeyPair pair = g.generateKeyPair();
```

Using a KeyFactory

With Explicit Parameters

As can be seen in the following code, the explicit parameters case for JDK 1.5 follows the same steps as for the Bouncy Castle provider as can be seen in the following code:

```
import java.security.spec.EllipticCurve;
import java.security.spec.ECFieldFp;
import java.security.spec.ECPoint;
import java.security.spec.ECParameterSpec;
import org.bouncycastle.jce.spec.ECPointUtil;
....
EllipticCurve curve = new EllipticCurve(
            new ECFieldFp(new
BigInteger("8834235323891921647916487503603088853144765972529603627924508606096998 3")), // q
            new
BigInteger("7fffffffffffffffffffff7fffffffffff8000000000007ffffffffffc", 16), //
a
            new
BigInteger("6b016c3bdcf18941d0d654921475ca71a9db2fb27d1d37796185c2942c0a", 16)); //
b
ECParameterSpec spec = new ECParameterSpec(
            curve,
            ECPointUtil.decodePoint(curve,
Hex.decode("020ffa963cdca8816ccc33b8642bedf905c3d358573d3f27fbbd3b3cb9aaaf")), // G
            new
BigInteger("8834235323891921647916487503603088480755034169162775227534542470280730 7"), // n
            1); // h

ECPrivateKeySpec priKey = new ECPrivateKeySpec(
            new
BigInteger("8763001015071075675010661307616710783570106710677817767166716761787267 17"), // d
            spec);
ECPublicKeySpec pubKey = new ECPublicKeySpec(
            ECPointUtil.decodePoint(curve,
Hex.decode("025b6dc53bc61a2548ffb0f671472de6c9521a9d2d2534e65abfcbd5fe0c70")), // Q
            spec);
```

The one difference of note is the use of the `ECPointUtil` class to handle an encoded point. The is a Bouncy Castle specific class which can be used to convert point encodings into JDK ECPoint objects. In the case where the point would have been added from its base `BigInteger` objects the following code could replace the call the `ECPointUtil`:

```
new ECPoint(
            new
BigInteger("5b6dc53bc61a2548ffb0f671472de6c9521a9d2d2534e65abfcbd5fe0c70", 16),
            new
BigInteger("7fd9f1ed2e65f09f6ce0893baf5e8e31e6ae82ea8c3592335be906d38dee", 16)),
```

With Named Curves

This case isn't actually directly supported in the JDK. Bouncy Castle does provide a helper class `org.bouncycastle.jce.spec.ECNamedCurveSpec` which can be used to wrap the return value from the named curve tables provided in `org.bouncycastle.jce`:

```
import java.security.spec.ECParameterSpec;
import java.security.spec.ECPrivateKeySpec;
import java.security.spec.ECPublicKeySpec;
import org.bouncycastle.jce.spec.ECNamedCurveSpec;
import org.bouncycastle.jce.spec.ECPointUtil;
...
ECParameterSpec params = new
ECNamedCurveSpec(ECNamedCurveTable.getParameterSpec("prime239v1"));
ECPrivateKeySpec priKey = new ECPrivateKeySpec(
                new
BigInteger("876300101507107567501066130761671078357010671067781776716671676178726717"), // d
                spec);
ECPublicKeySpec pubKey = new ECPublicKeySpec(
                     ECPointUtil.decodePoint(
                             params.getCurve(),

Hex.decode("025b6dc53bc61a2548ffb0f671472de6c9521a9d2d2534e65abfcbd5fe0c70")), // Q
                     spec);
```