#----------------------------------------------------------------------------------

#Rutvik Parmar

#H-41

#Date – 21st April 2016

#Assignement No:10 – ECC implementation

#----------------------------------------------------------------------------------

**public class** EllipticCurve {  
 *// The parameters of an EC.* **private** BigInteger **p**;  
 **private** BigInteger **a**;  
 **private** BigInteger **b**;  
  
 **public** EllipticCurve(BigInteger prime, BigInteger myA, BigInteger myB) {  
 **p** = prime;  
 **a** = myA;  
 **b** = myB;  
 }  
  
 *// Copy constructor.* **public** EllipticCurve(EllipticCurve copy) {  
 **p** = **new** BigInteger(copy.**p**.toString());  
 **a** = **new** BigInteger(copy.**a**.toString());  
 **b** = **new** BigInteger(copy.**b**.toString());  
 }  
  
 *// All three components must be equal for the curves to be the same.* **public boolean** equals(EllipticCurve other) {  
 **return p**.equals(other.**p**) && **a**.equals(other.**a**) && **b**.equals(other.**b**);  
 }  
  
 **public** BigInteger getP() {  
 **return p**;  
 }  
  
 **public** BigInteger getA() {  
 **return a**;  
 }  
}

**import** java.math.BigInteger;  
**public class** Point {  
 *// Store the x, y and curve.* **private** BigInteger **x**;  
 **private** BigInteger **y**;  
 **private** EllipticCurve **curve**;  
  
 *// Precondition: (myX, myY) must lie on the curve c. I don't check that here!!!* **public** Point(EllipticCurve c, BigInteger myX, BigInteger myY) {  
 **x** = myX;  
 **y** = myY;  
 **curve** = c;  
 }  
  
 *// Copy constructor.* **public** Point(Point copy) {  
 **x** = **new** BigInteger(copy.**x**.toString());  
 **y** = **new** BigInteger(copy.**y**.toString());  
 **curve** = **new** EllipticCurve(copy.**curve**);  
 }  
  
 *// Returns 0. Not sure if this is the proper way to store the "origin".* **public** Point(EllipticCurve c) {  
 **curve** = c;  
 **x** = BigInteger.***ZERO***;  
 **y** = BigInteger.***ZERO***;  
 }  
  
 *// All components must be equal...* **public boolean** equals(Point other) {  
 **return x**.equals(other.**x**) && **y**.equals(other.**y**) && **curve**.equals(other.**curve**);  
 }  
  
 *// Returns true iff other is this point's reflection over the line y = p/2 (real division)* **public boolean** mirror(Point other) {  
 **return x**.equals(other.**x**) && **curve**.equals(other.**curve**) && **y**.equals(other.**curve**.getP().subtract(other.**y**));  
 }  
  
 *// Returns the negative of this point, which is its mirror.* **public** Point negate() {  
  
 BigInteger newY = **curve**.getP().subtract(**y**);  
 **return new** Point(**curve**, **x**, newY);  
 }  
  
 *// Adds this to other and returns the answer, using the formulas in Stallings (5th edition)* **public** Point add(Point other) {  
  
 *// Can't add points on different curves.* **if** (!**curve**.equals(other.**curve**)) **return null**;  
  
 **if** (**this**.equals(other)) {  
  
 *// We need these to calculate lambda.* BigInteger three = **new** BigInteger(**"3"**);  
 BigInteger two = **new** BigInteger(**"2"**);  
 BigInteger temp = **new** BigInteger(**x**.toString());  
  
 *// Splitting up the calculation of lambda into all of these steps...* BigInteger lambda = temp.modPow(two, **curve**.getP());  
 lambda = three.multiply(lambda);  
 lambda = lambda.add(**curve**.getA());  
 BigInteger den = two.multiply(**y**);  
 lambda = lambda.multiply(den.modInverse(**curve**.getP()));  
  
 *// Once we have lambda, just plug into these equations.* BigInteger newX = lambda.multiply(lambda).subtract(**x**).subtract(**x**).mod(**curve**.getP());  
 BigInteger newY = (lambda.multiply(**x**.subtract(newX))).subtract(**y**).mod(**curve**.getP());  
 **return new** Point(**curve**, newX, newY);  
  
 }  
  
 *// Returns the origin...not sure if my origin is correct.* **else if** (**this**.mirror(other)) {  
 **return new** Point(**curve**);  
 }  
  
 *// Standard case.* **else** {  
  
 *// We need these to calculate lambda.* BigInteger three = **new** BigInteger(**"3"**);  
 BigInteger two = **new** BigInteger(**"2"**);  
 BigInteger temp = **new** BigInteger(**x**.toString());  
  
 *// Lambda's a bit easier here...* BigInteger lambda = other.**y**.subtract(**y**);  
 BigInteger den = other.**x**.subtract(**x**);  
 lambda = lambda.multiply(den.modInverse(**curve**.getP()));  
  
 *// This calculation is roughly the same as above.* BigInteger newX = lambda.multiply(lambda).subtract(**x**).subtract(other.**x**).mod(**curve**.getP());  
 BigInteger newY = (lambda.multiply(**x**.subtract(newX))).subtract(**y**).mod(**curve**.getP());  
 **return new** Point(**curve**, newX, newY);  
  
 }  
 }  
  
 *// Subtraction is just adding the negative.* **public** Point subtract(Point other) {  
 other = other.negate();  
 **return this**.add(other);  
 }  
  
 *// Uses "fast multiplication" to multiply this point by factor.* **public** Point multiply(BigInteger factor) {  
  
 BigInteger two = **new** BigInteger(**"2"**);  
  
 *// Base cases.* **if** (factor.equals(BigInteger.***ONE***))  
 **return new** Point(**this**);  
 **if** (factor.equals(two))  
 **return this**.add(**this**);  
  
 *// Even case where we can calculate half of our answer and multiply by 2!* **if** (factor.mod(two).equals(BigInteger.***ZERO***)) {  
 Point sqrt = multiply(factor.divide(two));  
 **return** sqrt.add(sqrt);  
 }  
  
 *// No speed up here, but this recursive call will lead to one.* **else** {  
 factor = factor.subtract(BigInteger.***ONE***);  
 **return this**.add(multiply(factor));  
 }  
  
 }  
  
 **public** String toString() {  
 **return "("** + **x** +**", "**+**y**+**")"**;  
 }  
  
 **public static void** main(String[] args) {  
  
 *// Test out Point arithmetic.* EllipticCurve myCurve = **new** EllipticCurve(**new** BigInteger(**"23"**), **new** BigInteger(**"1"**), **new** BigInteger(**"1"**));  
 Point p = **new** Point(myCurve, **new** BigInteger(**"3"**), **new** BigInteger(**"10"**));  
 Point q = **new** Point(myCurve, **new** BigInteger(**"9"**), **new** BigInteger(**"7"**));  
  
 *// P + Q* Point pPlusq = p.add(q);  
 System.***out***.println(p+**" + "**+q+**" = "**+pPlusq);  
  
 *// 2P* Point twoP = p.add(p);  
 System.***out***.println(**"2\* "**+p+**" = "**+twoP);  
  
 *// Test multiplication =)* Point fourP = p.multiply(**new** BigInteger(**"4"**));  
 System.***out***.println(**"Four times p = "**+fourP);  
  
 Point checkFourP = twoP.add(twoP);  
 System.***out***.println(**"check Four times p = "**+checkFourP);  
  
 }  
}

**import** java.math.\*;  
**import** java.util.\*;  
  
**public class** ECC {  
  
 *// Parts of one ECC system.* **private** EllipticCurve **curve**;  
 **private** Point **generator**;  
 **private** Point **publicKey**;  
 **private** BigInteger **privateKey**;  
  
 *// We need a curve, a generator point (x,y) and a private key, nA, that will  
 // be used to generate the public key.* **public** ECC(EllipticCurve c, BigInteger x, BigInteger y, BigInteger nA) {  
  
 **curve** = c;  
 **generator** = **new** Point(**curve**, x, y);  
 **privateKey** = nA;  
 **publicKey** = **generator**.multiply(**privateKey**);  
 }  
  
 *// Encryption.* **public** Point[] encrypt(Point plain) {  
  
 *// First we must pick a random k, in range.* **int** bits = **curve**.getP().bitLength();  
 BigInteger k = **new** BigInteger(bits, **new** Random());  
 System.***out***.println(**"Picked "**+k+**" as k for encrypting."**);  
  
 *// Our output is an ordered pair, (k\*generator, plain + k\*publickey)* Point[] ans = **new** Point[2];  
 ans[0] = **generator**.multiply(k);  
 ans[1] = plain.add(**publicKey**.multiply(k));  
 **return** ans;  
 }  
  
 *// Decryption - notice the similarity to El Gamal!!!* **public** Point decrypt(Point[] cipher) {  
  
 *// This is what we subtract out.* Point sub = cipher[0].multiply(**privateKey**);  
  
 *// Subtract out and return.* **return** cipher[1].subtract(sub);  
 }  
  
 **public** String toString() {  
  
 **return "Gen: "**+**generator**+**"\n"**+  
 **"pri: "**+**privateKey**+**"\n"**+  
 **"pub: "**+**publicKey**;  
 }  
  
 **public static void** main(String[] args) {  
  
  
 *// Just use the book's curve and test.* EllipticCurve myCurve = **new** EllipticCurve(**new** BigInteger(**"23"**), **new** BigInteger(**"1"**), **new** BigInteger(**"1"**));  
 BigInteger x = **new** BigInteger(**"6"**);  
 BigInteger y = **new** BigInteger(**"19"**);  
 BigInteger nA = **new** BigInteger(**"10"**);  
 ECC Alice = **new** ECC(myCurve, x, y, nA);  
  
 *// I have hard-coded my plaintext point.* Point plain = **new** Point(myCurve, **new** BigInteger(**"3"**), **new** BigInteger(**"13"**));  
 System.***out***.println(**"encrypting "**+plain);  
  
 *// Encrypt and print.* Point[] cipher = Alice.encrypt(plain);  
 System.***out***.println(**"cipher first part "**+cipher[0]);  
 System.***out***.println(**"cipher second part "**+cipher[1]);  
  
 *// Decrypt and verify.* Point recover = Alice.decrypt(cipher);  
 System.***out***.println(**"recovered "**+recover);  
  
 }  
}

Output –

encrypting (3, 13)

Picked 9 as k for encrypting.

cipher first part (12, 19)

cipher second part (9, 7)

recovered (3, 13)