```java
import java.util.Random;

public class Main {
    public static void main(String[] args) {
        // the lower and upper bounds for the prime number
        int min = 10000;
        int max = 100000;
        int prime;
        int base;
        //this calls the genPrime method to generate a random number until the isPrime function returns true and checks that it's a prime number
        do {
            prime = genPrime(min, max);
        }
        while (!isPrime(prime));

        System.out.println("\nPrime number = " + prime);

        //base is set to the value of a primitive root of the prime number
        base = findPrimRoot(prime);

        System.out.println("\nPrimitive root = " + base);

        //private keys are generated for alice and bob
        int alice = genPriv(prime);
        System.out.println("\nPrivate key for alice = " + alice);
        int bob = genPriv(prime);
        System.out.println("\nPrivate key for bob = " + bob);

        //public keys are generated for alice and bob
        int x = calcPub(base, alice, prime);
        System.out.println("\nPublic key for alice =" + x);
        int y = calcPub(base, bob, prime);
        System.out.println("\nPublic key for bob =" + y);

        //and finally the secret keys are generated for alice and bob, using eachother's shared public keys. The secret keys should be equal
        int ka = calcSec(y, alice, prime);
        System.out.println("\nSecret key for alice =" + ka);
        int kb = calcSec(x, bob, prime);
        System.out.println("\nSecret key for bob =" + kb);

        //part 2 mallory mim attack
        int Mallory = genPriv(prime);

        System.out.println("\nPrivate key for mallory = " + Mallory);

        int pubMallory = calcPub(base, Mallory, prime);

        System.out.println("\nPublic key for mallory = " + pubMallory);

        //bob and alice send their public keys but mallory intercepts and sends her own public key
        int MallInterceptAliceKey = calcSec(x, Mallory, prime);
        int AliceKey = calcSec(pubMallory, alice, prime);
        int MallInterceptBobKey = calcSec(y, Mallory, prime);
        int BobKey = calcSec(pubMallory, bob, prime);

        System.out.println("\nAlice sent bob public key " + x + ", Mallory intercepted it and generated the secret key " + MallInterceptAliceKey + ", and sent bob the secret key");
```

```
        System.out.println("\nBob sent alice public key " + y + ", Mallory
intercepted it and generated the secret key " + MallInterceptBobKey + ",
and sent bob the secret key");
    }

    //to generate a random(hopefully prime) number
    public static int genPrime(int min, int max) {
        int p = 0;
        Random random = new Random();
        // random number is generated between the lower and upper bounds
        p = random.nextInt((max - min) + 1) + min;
        return p;
    }

    //to check if the prime number generated is in fact a prime number
    public static boolean isPrime(int p) {
        if (p <= 1) {
            return false; // a prime number must be bigger than +1
        }
        //we only iterate the divisor up to the square root of the prime
number to save time as the product of 2 numbers bigger than the square root
would be larger than the prime number itself
        for (int i = 2; i <= Math.sqrt(p); i++) {
            if (p % i == 0) {
                return false; //divisor mustn't divide equally into the
prime number as the prime number can only have 2 divisors, 1 and itself
            }
        }
        return true;
    }

    //method to iterate through numbers until it finds a primitive root
    public static int findPrimRoot(int p) {
        for(int a = 2; a < p; a++) { //it starts at 2 and iterates up to
prime - 1
            boolean primRoot = true;
            for(int i = 1; i < p; i++) {
                if(modP(a, i, p) == 0) { //checks if a to the power of i
modulo prime number gives 0, if it does we can straight away say that a is
not a primitive root
                    primRoot = false;
                    break;
                }
            }
            if (primRoot) {
                return a; //returns the primitive root
            }
        }
        return -1;
    }

//method that generates a private key less than the prime number
    public static int genPriv(int p) {
        Random random = new Random();

        return random.nextInt(p - 1) + 1;
    }
//method that uses the modP() function to calculate public key
    public static int calcPub(int base, int privKey, int prime) {
        return modP(base, privKey, prime);
    }
```

```
    //method that uses modP() to calculate the secret key
    public static int calcSec(int pubKey, int privKey, int prime) {
        return modP(pubKey, privKey, prime);
    }


    public static int modP(int base, int exponent, int modulus) {
        if (modulus == 1) {
            return 0; //prime can't be equal to 1
        }

        int result = 1;

        base = base % modulus; // base is reduced modulo the modulus to
ensure it's within the range 0 - modulo-1

        while (exponent > 0) {
            if (exponent % 2 == 1) {
                result = (result * base) % modulus; //this loop breaks up
and iterates through the exponent by dividing it into 2, it multiplies the
result by the current base value and takes the result modulo the modulus,
and then shifts the exponent to it's next bit, this breaks up the
calculations and saves time
            }
            exponent = exponent >> 1;
            base = (base * base) % modulus; // base is squared modulo the
modulus.
        }
        return result; //returns the result base^exponent % modulus
    }
}
```