p1q9

2000

ACN

more restrictive in access than in the parent. Etc Etc!!
The "//* ... */" comment at the head is valid but looks
wrong. The print method (such as it is) would bard when
it hits a null reference. LOTS for people to spot and
a proper marking scheme grades in part on which cases
turn out in reality to be easy for people to spot and
on clarity of explanation.

2 Java full

Write Java classs that provides support for arithmetic on the integers worked with relative to some prime modulus p. An instance of the class should be constuctable specifying the modulus, and then it should provide methods to create numbers and add, subtract, multiply, divide and print them. Note that the Discrete Mathematics lectures explained about arithmetic modulo a prime, and that the reciprocal of a number (a say) (mod p) can be found by solving the equation

$$ab = 1 \bmod p$$

As a sample of the desired behaviour for your class, here is some test code for it:

```
Modular d = new Modular(7);  // work mod 7
ModInt a = d.reduceMod(10);  // create "10 mod 7"
ModInt b = d.reduceMod(20);  // create "20 mod 7"
ModInt c = a.divide(b);      // work out a/b mod 7
c.print();
```

Note that I am suggesting a class called `Modular` that keeps track of the modulus p, and a second class ModInt to stand for numbers: these are created for the user via a method in `Modular`

Your code should compain in some manner if an attempt is made to (say) add a number that is defined modula 7 to one that is defined modulo 11.

[20 marks]

2.1 Marking notes

```
class Modular
{
int p;
Modular(int p)
{ this.p = p; }

ModInt reduceMod(int n)
{
    return new ModInt(n%p, p);
}
}

class ModInt
{
```

```
int n, p;
ModInt(int n, int p)
{   this.n = n; this.p = p;
}

ModInt add(ModInt b)
{
    if (p != b.p) <moan in some way>
    return new ModInt((a+b)%p, p);
}
// etc for subtract, multiply

Division code must implement extended Euclidean GCD as per
discrete maths notes!
}
```