

Convert this FORTRAN program to any one of the programming languages listed above. Submit your code and a screen shot of the output (together in one file) to Canvas as a PDF or docx or txt.

```

program conpay

  integer year
  real owed, paymnt, rate, x, round2, next
  data owed/10000.0/, paymnt /1000.0/, rate /8.0/
  round2(x) = real(NINT(x*100.0))/ 100.0
  next(x) = x* (1.0 + rate /100.0)

  print *, '          year   owed before      payment   owed after'
  print *, '          payment                payment'
  print *, '-----'

  year = 1
11  if (paymnt .le. next(owed)) then
    owed = next(owed)
    owed = round2(owed)
    print *, year, owed, paymnt, owed - paymnt

    year = year + 1
    owed = owed - paymnt
    go to 11
  end if

  owed = next(owed)
  owed = round2(owed)

  print *, year, owed, owed, 0.0
  print *, '-----'
end

```

Solution:

```

import java.math.BigDecimal;
import java.math.RoundingMode;
public class Main {
    public static void main(String[] args) {
        BigDecimal owed = new BigDecimal("10000.0000");
        BigDecimal payment = new BigDecimal("1000.0000");
        BigDecimal rate = new BigDecimal("8.0000");

        System.out.println("          year   owed before      payment   owed
after");
        System.out.println("          payment                payment");
        System.out.println("-----");

        int year = 1;
        while (payment.compareTo(next(owed, rate)) <= 0) {
            owed = next(owed, rate);
            owed = round2(owed);
            System.out.printf("%11d %13.7f%17.5f%17.5f%n", year, owed,
payment, owed.subtract(payment));
            year++;
            owed = owed.subtract(payment);
        }
    }
}

```

```

    }

    owed = next(owed, rate);
    owed = round2(owed);
    System.out.printf("%11d%13.6f%17.6f%17.8f%n", year, owed, owed,
BigDecimal.ZERO);

System.out.println("-----");
}

private static BigDecimal round2(BigDecimal value) {
    return value.setScale(2, RoundingMode.HALF_UP);
}

private static BigDecimal next(BigDecimal value, BigDecimal rate) {

    BigDecimal hundred = new BigDecimal("100");

    BigDecimal rateDivided = rate.divide(hundred, 20,
RoundingMode.HALF_EVEN);

    BigDecimal increment = BigDecimal.ONE.add(rateDivided);

    BigDecimal result = value.multiply(increment).setScale(10,
RoundingMode.HALF_EVEN);

    result = result.setScale(4, RoundingMode.HALF_EVEN);
    return result;
}
}

```

Note on Precision Discrepancies Between Java and FORTRAN:

The small difference in precision between my Java implementation and the original FORTRAN program (e.g., 10350.7200 in Java versus 10350.7197 in FORTRAN) occurs due to how the two languages handle floating-point arithmetic and rounding.

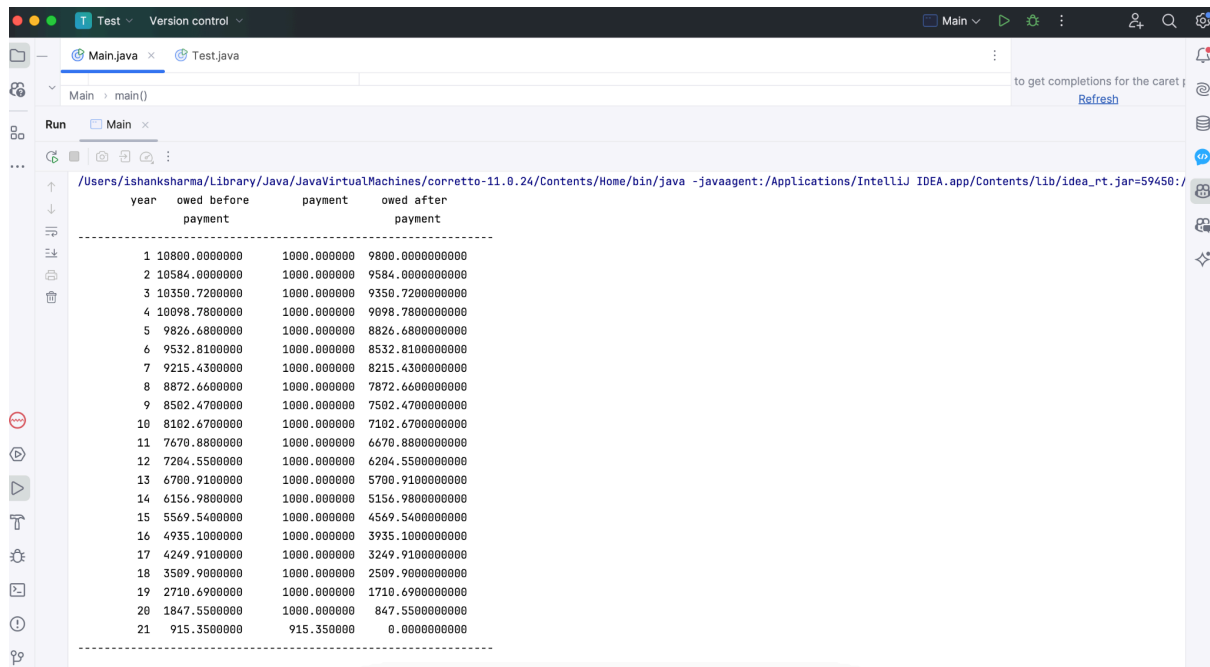
Rounding Mechanisms: In FORTRAN, the NINT function rounds values to the nearest integer, which behaves slightly differently than Java's RoundingMode.HALF_EVEN. While FORTRAN's NINT rounds halfway cases in one way, Java's BigDecimal might round those same values differently, resulting in outcomes like 10350.7200 instead of 10350.7197.

Floating-Point Arithmetic: I used BigDecimal in Java to ensure precision for financial calculations. However, due to differences in how FORTRAN and Java handles floating-point arithmetic cause slight variations. As per my understanding Java's BigDecimal enforces strict precision and rounding rules, whereas FORTRAN might handle real numbers more flexibly.

Expected Output:

year	owed before payment	payment	owed after payment
1	10800.0000	1000.00000	9800.00000
2	10584.0000	1000.00000	9584.00000
3	10350.7197	1000.00000	9350.71973
4	10098.7803	1000.00000	9098.78027
5	9826.67969	1000.00000	8826.67969
6	9532.80957	1000.00000	8532.80957
7	9215.42969	1000.00000	8215.42969
8	8872.66016	1000.00000	7872.66016
9	8502.46973	1000.00000	7502.46973
10	8102.66992	1000.00000	7102.66992
11	7670.87988	1000.00000	6670.87988
12	7204.54980	1000.00000	6204.54980
13	6700.91016	1000.00000	5700.91016
14	6156.97998	1000.00000	5156.97998
15	5569.54004	1000.00000	4569.54004
16	4935.10010	1000.00000	3935.10010
17	4249.91016	1000.00000	3249.91016
18	3509.89990	1000.00000	2509.89990
19	2710.68994	1000.00000	1710.68994
20	1847.55005	1000.00000	847.550049
21	915.349976	915.349976	0.00000000

My Output:



```
Test - Version control
Main.java x Test.java
Main -> main()
Run Main x
/Users/ishanksharma/Library/Java/JavaVirtualMachines/corretto-11.0.24/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA.app/Contents/lib/idea_rt.jar=59450:/
year owed before payment owed after
-----
1 10800.000000 1000.000000 9800.000000000
2 10584.000000 1000.000000 9584.000000000
3 10350.720000 1000.000000 9350.720000000
4 10098.780000 1000.000000 9098.780000000
5 9826.680000 1000.000000 8826.680000000
6 9532.810000 1000.000000 8532.810000000
7 9215.430000 1000.000000 8215.430000000
8 8872.660000 1000.000000 7872.660000000
9 8502.470000 1000.000000 7502.470000000
10 8102.670000 1000.000000 7102.670000000
11 7670.880000 1000.000000 6670.880000000
12 7204.550000 1000.000000 6204.550000000
13 6700.910000 1000.000000 5700.910000000
14 6156.980000 1000.000000 5156.980000000
15 5569.540000 1000.000000 4569.540000000
16 4935.100000 1000.000000 3935.100000000
17 4249.910000 1000.000000 3249.910000000
18 3509.900000 1000.000000 2509.900000000
19 2710.690000 1000.000000 1710.690000000
20 1847.550000 1000.000000 847.550000000
21 915.350000 915.350000 0.000000000
-----
```